Lecture Notes in Artificial Intelligence 1866
Subseries of Lecture Notes in Computer Science

James Cussens Alan Frisch (Eds.)

Inductive Logic Programming

10th International Conference, ILP 2000
London, UK, July 2000
Proceedings

Springer

Lecture Notes in Artificial Intelligence 1866

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

James Cussens Alan Frisch (Eds.)

Inductive Logic Programming

10th International Conference, ILP 2000
London, UK, July 24-27, 2000
Proceedings

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saabrücken, Germany

Volume Editors

James Cussens
Alan Frisch
University of York, Department of Computer Science
Heslington, York YO10 5DD, UK
E-mail: {jc,frisch}@cs.york.ac.uk

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Inductive logic programming : 10th international conference ;
proceedings / ILP 2000, London, UK, July 24 - 27, 2000. James Cussens ;
Alan Frisch (ed.) - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1866 : Lecture notes in
artificial intelligence)
ISBN 3-540-67795-X

CR Subject Classification (1998): I.2, D.1.6

ISBN 3-540-67795-X Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Boller Mediendesign
Printed on acid-free paper SPIN: 10722272 06/3142 5 4 3 2 1 0

Foreword

This volume contains one invited and fifteen submitted papers presented at the Tenth International Conference on Inductive Logic Programming (ILP 2000). The fifteen accepted papers were selected by the program committee from the 37 papers submitted to the conference. Each paper was carefully reviewed by three referees.

ILP 2000 was held at Imperial College, London, 24-27 July 2000 and was integrated with the First International Conference on Computational Logic (CL 2000). With ILP's strong roots in computational logic, this was a natural marriage. CL 2000 was a five-day extravaganza, incorporating both the Sixth International Conference on Rules and Objects in Databases (DOOD 2000) and the Tenth International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR 2000) and featuring eight invited speakers, twelve tutorials, and seven affiliated workshops. Registrants for CL 2000 and ILP 2000 could move freely between the two events, the main distinction between the events being separate conference proceedings.

We wish to thank all the authors who submitted their papers to ILP 2000; the program committee members, and other reviewers who did a thorough job in spite of demanding deadlines; and our invited speaker, David Page. Thanks also to Alfred Hofmann, and everyone else at Springer for their smooth handling of these proceedings. We would also like to thank the organisers of CL 2000, whose cooperation brought the two events together: John Lloyd (Program Chair), Marek Sergot (Conference Chair), Frank Kriwaczek and Francesca Toni (Local Organisers), Femke van Raamsdonk (Publicity Chair) and Sandro Etalle (Workshop Chair). Finally, we are grateful to our sponsors for their financial support.

May 2000

James Cussens & Alan Frisch
Program Chairs
ILP 2000

ILP 2000 Program Committee

Henrik Boström (University of Stockholm, Sweden)
Ivan Bratko (University of Ljubljana, Slovenia)
James Cussens (University of York, UK)
Shan-Hwei Nienhuys-Cheng (University of Rotterdam, Netherlands)
William Cohen (Whizbangs Labs, USA)
Luc DeRaedt (University of Freiburg, Germany)
Sašo Džeroski (Jožef Stefan Institute, Ljubljana)
Peter Flach (University of Bristol, UK)
Alan Frisch (University of York, UK)
Koichi Furukawa (University of Keio, Japan)
Roni Khardon (University of Edinburgh, UK)
Jörg-Uwe Kietz (Swiss Life, Switzerland)
Nada Lavrač (Jožef Stefan Institute, Slovenia)
John Lloyd (Australian National University, Australia)
Stan Matwin (University of Ottawa, Canada)
Raymond Mooney (University of Texas, USA)
Stephen Muggleton (University of York, UK)
David Page (University of Wisconsin, USA)
Bernhard Pfahringer (University of Waikato, New Zealand)
Céline Rouveirol (Université de Paris-Sud, France)
Claude Sammut (University of New South Wales, Australia)
Michèle Sebag (École Polytechnique, France)
Ashwin Srinivasan (University of Oxford, UK)
Prasad Tadepalli (Oregon State University, USA)
Stefan Wrobel (University of Magdeburg, Germany)
Akihiro Yamamoto (University of Hokkaido, Japan)

Additional Referees

Érick Alphonse (Université de Paris-Sud, France)
Liviu Badea (National Institute for Research and Development in Informatics, Romania)
Damjan Demsar (Jožef Stefan Institute, Slovenia)
Elisabeth Goncalves (Université de Paris-Sud, France)
Marko Grobelnik (Jožef Stefan Institute, Slovenia)
Claire Kennedy (University of Bristol, UK)
Daniel Kudenko (University of York, UK)
Johanne Morin (University of Ottawa, Canada)
Tomonobu Ozaki (Keio University, Japan)
Edward Ross (University of Bristol, UK)
Ljupco Todorovski (Jožef Stefan Institute, Slovenia)
Véronique Ventos (Université de Paris-Sud, France)

Sponsors of ILP 2000

ILPNet2, The European Network of Excellence in Inductive Logic Programming
MLNet, The European Network of Excellence in Machine Learning
CompulogNet, The European Network of Excellence in Computational Logic

Table of Contents

ILP: Just Do It

David Page

Dept. of Biostatistics and Medical Informatics
and Dept. of Computer Sciences
University of Wisconsin
1300 University Ave., Rm 5795 Medical Sciences
Madison, WI 53706
U.S.A.
page@biostat.wisc.edu

Abstract. Inductive logic programming (ILP) is built on a foundation laid by research in other areas of computational logic. But in spite of this strong foundation, at 10 years of age ILP now faces a number of new challenges brought on by exciting application opportunities. The purpose of this paper is to interest researchers from other areas of computational logic in contributing their special skill sets to help ILP meet these challenges. The paper presents five future research directions for ILP and points to initial approaches or results where they exist. It is hoped that the paper will motivate researchers from throughout computational logic to invest some time into "doing" ILP.

1 Introduction

Inductive Logic Programming has its foundations in computational logic, including logic programming, knowledge representation and reasoning, and automated theorem proving. These foundations go well beyond the obvious basis in definite clause logic and SLD-resolution. In addition ILP has heavily utilized such theoretical results from computational logic as Lee's Subsumption Theorem [18], Gottlob's Lemma linking implication and subsumption [12], Marcinkowski and Pacholski's result on the undecidability of implication between definite clauses [22], and many others. In addition to utilizing such theoretical results, ILP depends crucially on important advances in logic programming implementations. For example, many of the applications summarized in the next brief section were possible only because of fast deductive inference based on indexing, partial compilation, etc. as embodied in the best current Prolog implementations. Furthermore, research in computational logic has yielded numerous important lessons about the art of knowledge representation in logic that have formed the basis for applications. Just as one example, definite clause grammars are central to several ILP applications within both natural language processing and bioinformatics.

ILP researchers fully appreciate the debt we owe to the rest of computational logic, and we are grateful for the foundation that computational logic has provided. Nevertheless, the goal of this paper is not merely to express gratitude, but

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 3–18, 2000.

also to point to the present and future needs of ILP research. More specifically, the goal is to lay out future directions for ILP research and to attract researchers from the various other areas of computational logic to contribute their unique skill sets to some of the challenges that ILP now faces.[1] In order to discuss these new challenges, it is necessary to first briefly survey some of the most challenging application domains of the future. Section 2 provides such a review. Based on this review, Section 3 details five important research directions and concomitant challenges for ILP, and Section 4 tries to "close the sale" in terms of attracting new researchers.

2 A Brief Review of Some Application Areas

One of the most important application domains for machine learning in general is bioinformatics, broadly interpreted. This domain is particularly attractive for (1) its obvious importance to society, and (2) the plethora of large and growing data sets. Data sets obviously include the newly completed and available DNA sequences for *C. elegans* (nematode), *Drosophila* (fruitfly), and (depending on one's definitions of "completed" and "available") man. But other data sets include gene expression data (recording the degree to which various genes are expressed as protein in a tissue sample), bio-activity data on potential drug molecules, x-ray crystallography and NMR data on protein structure, and many others. Bioinformatics has been a particularly strong application area for ILP, dating back to the start of Stephen Muggleton's collaborations with Mike Sternberg and Ross King [29, 16]. Application areas include protein structure prediction [29, 37], mutagenicity prediction [17], and pharmacophore discovery [7] (discovery of a 3D substructure responsible for drug activity that can be used to guide the search for new drugs with similar activity). ILP is particularly well-suited for bioinformatics tasks because of its abilities to take into account background knowledge and structured data and to produce human-comprehensible results. For example, the following is a potential pharmacophore for ACE inhibition (a form of hypertension medication), where the spacial relationships are described through pairwise distances.[2]

```
Molecule A is an ACE inhibitor if:
   molecule A contains a zinc binding site B, and
   molecule A contains a hydrogen acceptor C, and
   the distance between B and C is 7.9 +/- .75 Angstroms, and
   molecule A contains a hydrogen acceptor D, and
   the distance between B and D is 8.5 +/- .75 Angstroms, and
   the distance between C and D is 2.1 +/- .75 Angstroms, and
   molecule A contains a hydrogen acceptor E, and
   the distance between B and E is 4.9 +/- .75 Angstroms, and
```

[1] Not to put too fine a point on the matter, this paper contains unapologetic proselytizing.

[2] Hydrogen acceptors are atoms with a weak negative charge. Ordinarily, zinc-binding would be irrelevant; it is relevant here because ACE is one of several proteins in the body that typically contains an associated zinc ion. This is an automatically generated translation of an ILP-generated clause.

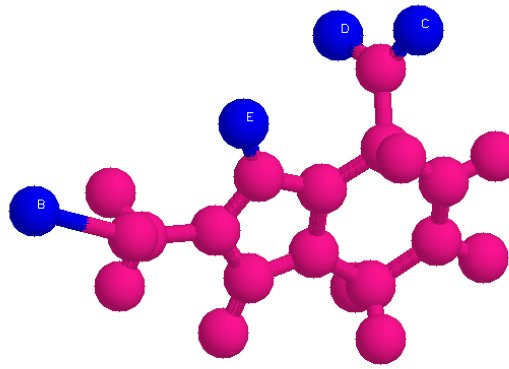


Fig. 1. ACE inhibitor number 1 with highlighted 4-point pharmacophore.

```
the distance between C and E is 3.1 +/- .75 Angstroms, and
the distance between D and E is 3.8 +/- .75 Angstroms.
```

Figures 1 and 2 show two different ACE inhibitors with the parts of pharmacophore highlighted and labeled.

A very different type of domain for machine learning is natural language processing (NLP). This domain also includes a wide variety of tasks such as part-of-speech tagging, grammar learning, information retrieval, and information extraction. Arguably, natural language translation (at least, very rough-cut translation) is now a reality—witness for example the widespread use of Altavista's Babelfish. Machine learning techniques are aiding in the construction of information extraction engines that fill database entries from document abstracts (e.g., [3]) and from web pages (e.g., WhizBang! Labs, http://www.whizbanglabs.com). NLP became a major application focus for ILP in particular with the ESPRIT project ILP^2. Indeed, as early as 1998 the majority of the application papers at the ILP conference were on NLP tasks.

A third popular and challenging application area for machine learning is knowledge discovery from large databases with rich data formats, which might contain for example satellite images, audio recordings, movie files, etc. While Dzeroski has shown how ILP applies very naturally to knowledge discovery from ordinary relational databases [6], advances are needed to deal with multimedia databases.

ILP has advantages over other machine learning techniques for all of the preceding application areas. Nevertheless, these and other potential applications also highlight the following shortcomings of present ILP technology.

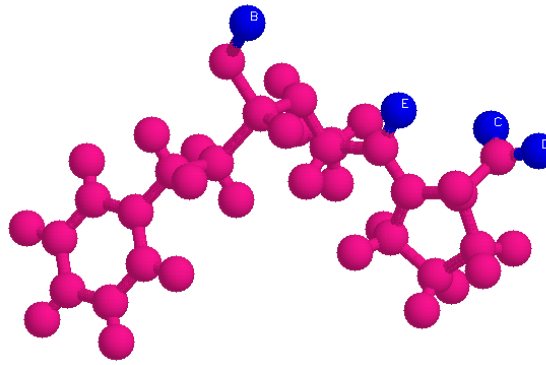


Fig. 2. ACE inhibitor number 2 with highlighted 4-point pharmacophore.

- Other techniques such as hidden Markov models, Bayes Nets and Dynamic Bayes Nets, and bigrams and trigrams can expressly represent the probabilities inherent in tasks such as part-of-speech tagging, alignment of proteins, robot maneuvering, etc. Few ILP systems are capable of representing or processing probabilities.[3]
- ILP systems have higher time and space requirements than other machine learning systems, making it difficult to apply them to large data sets. Alternative approaches such as stochastic search and parallel processing need to be explored.
- ILP works well when data and background knowledge are cleanly expressible in first-order logic. But what can be done when databases contain images, audio, movies, etc.? ILP needs to learn lessons from constraint logic programming regarding the incorporation of special-purpose techniques for handling special data formats.
- In scientific knowledge discovery, for example in the domain of bioinformatics, it would be beneficial if ILP systems could collaborate with scientists rather than merely running in batch mode. If ILP does not take this step, other forms of collaborative scientific assistants will be developed, supplanting ILP's position within these domains.

[3] It should be noted that Stephen Muggleton and James Cussens have been pushing for more attention to probabilities in ILP. Stephen Muggleton initiated this direction with an invited talk at ILP'95 and James Cussens has a recently-awarded British EPSRC project along these lines. Nevertheless, litte attention has been paid to this shortcoming by other ILP researchers, myself included.

In light of application domains and the issues they raise, the remainder of this paper discusses five directions for future research in ILP. Many of these directions require fresh insights from other areas of computational logic. The author's hope is that this discussion will prompt researchers from other areas to begin to explore ILP.[4]

3 Five Directions for ILP Research

Undoubtedly there are more than five important directions for ILP research. But five directions stand out clearly at this point in time. They stand out not only in the application areas just mentioned, but also when examining current trends in AI research generally. These areas are

- incorporating explicit probabilities into ILP
- stochastic search
- building special-purpose reasoners into ILP
- enhancing human-computer interaction to make ILP systems true *collaborators* with human experts
- parallel execution using commodity components

Each of these research directions can contribute substantially to the future widespread success of ILP. And each of these directions could benefit greatly from the expertise of researchers from other areas of computational logic. This section discusses these five research directions in greater detail.

3.1 Probabilistic Inference: ILP and Bayes Nets

Bayesian Networks have largely supplanted traditional rule-based expert systems. Why? Because in task after task we (AI practitioners) have realized that probabilities are central. For example, in medical diagnosis few universally true rules exist and few entirely accurate laboratory experiments are available. Instead, probabilities are needed to model the task's inherent uncertainty. Bayes Nets are designed specifically to model probability distributions and to reason about these distributions accurately and (in some cases) efficiently. Consequently, in many tasks including medical diagnosis [15], Bayes Nets have been found to be superior to rule-based systems. Interestingly, inductive inference, or machine learning, has turned out to be a very significant component of Bayes Net reasoning. Inductive inference from data is particularly important for developing or adjusting the conditional probability tables (CPTs) for various network nodes, but also is used in some cases even for developing or modifying the structure of the network itself.

[4] It is customary in technical papers for the author to refer to himself in the third person. But because the present paper is an invited paper expressing the author's opinions, the remainder will be much less clumsy if the author dispenses with that practice, which I now will do.

But not all is perfection and contentment in the world of Bayes Nets. A Bayes Net is less expressive than first-order logic, on a par with propositional logic instead. Consequently, while a Bayes Net is a graphical representation, it cannot represent relational structures. The only relationships captured by the graphs are conditional dependencies among probabilities. This failure to capture other relational information is particularly troublesome when using the Bayes Net representation in learning. For a concrete illustration, consider the task of pharmacophore discovery. It would be desirable to learn probabilistic predictors, e.g., what is the probability that a given structural change to the molecule fluoxetine (Prozac) will yield an equally effective anti-depressant (specifically, serotonin reuptake inhibitor)? To build such a probabilistic predictor, we might choose to learn a Bayes Net from data on serotonin reuptake inhibitors. Unfortunately, while a Bayes Net can capture the probabilistic information, it cannot capture the structural properties of a molecule that are predictive of biological activity.

The inability of Bayes Nets to capture relational structure is well known and has led to attempts to extend the Bayes Net representation [8, 9] and to study inductive learning with such an extended representation. But the resulting extended representations are complex and yet fall short of the expressivity of first-order logic. An interesting alternative for ILP researchers to examine is learning clauses with probabilities attached. It will be important in particular to examine how such representations and learning algorithms compare with the extended Bayes Net representations and learning algorithms. Several candidate clausal representations have been proposed and include probabilistic logic programs, stochastic logic programs, and probabilistic constraint logic programs; Cussens provides a nice survey of these representations [5]. Study already has begun into algorithms and applications for learning stochastic logic programs [27], and this is an exciting area for further work. In addition, the first-order representation closest to Bayes Nets is that of Ngo and Haddawy. The remainder of this subsection points to approaches for, and potential benefits of, learning these clauses in particular.

Clauses in the representation of Ngo and Haddawy may contain random variables as well as ordinary logical variables. A clause may contain at most one random variable in any one literal, and random variables may appear in body literals only if a random variable appears in the head. Finally, such a clause also has a Bayes Net fragment attached, which may be thought of as a constraint. This fragment has a very specific form. It is a directed graph of node depth two (edge depth one), with all the random variables from the clause body as parents of the random variable from the clause head.[5] Figure 3 provides an example of such a clause as might be learned in pharmacophore discovery (CPT not shown). This clause enables us to specify, through a CPT, how the probability of a molecule being active depends on the particular values assigned to the distance variables

[5] This is not exactly the definition provided by Ngo and Haddawy, but it is an equivalent one. Readers interested in deductive inference with this representation are encouraged to see [31, 30].

$D1$, $D2$, and $D3$. In general, the role of the added constraint in the form of a Bayes net fragment is to define a conditional probability distribution over the random variable in the head, conditional on the values of the random variables in the body. When multiple such clauses are chained together during inference, a larger Bayes Net is formed that defines a joint probability distribution over the random variables.

```
drug(Molecule,Activity_Level):-
    contains_hydrophobe(Molecule,Hydrophobe),
    contains_basic_nitrogen(Molecule,Nitrogen),
    contains_hydrogen_acceptor(Molecule,Acceptor),
    distance(Molecule,Hydrophobe,Nitrogen,D1),
    distance(Molecule,Hydrophobe,Acceptor,D2),
    distance(Molecule,Nitrogen,Acceptor,D3).
```

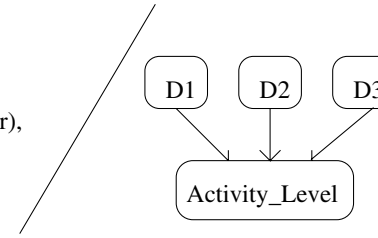


Fig. 3. A clause with a Bayes Net fragment attached (CPT not included). The random variables are *Activity_Level*, D1, D2, and D3. Rather than using a hard range in which the values of D1, D2, and D3 must fall, as the pharmacophores described earlier, this new representation allows us to describe a probability distribution over *Activity_Level* in terms of the values of D1, D2, and D3. For example, we might assign higher probabilities to high *Activity_Level* as D1 gets closer to 3 Angstroms from either above or below. The CPT itself might be a linear regression model, i.e. a linear function of D1, D2, and D3 with some fixed variance assumed, or it might be a discretized model, or other.

I conjecture that existing ILP algorithms can effectively learn clauses of this form with the following modification. For each clause constructed by the ILP algorithm, collect the positive examples covered by the clause. Each positive example provides a value for the random variable in the head of the clause, and because the example is covered, the example together with the background knowledge provides values for the random variables in the body. These values, over all the covered positive examples, can be used as the data for constructing the conditional probability table (CPT) that accompanies the attached Bayes Net fragment. When all the random variables are discrete, a simple, standard method exists for constructing CPTs from such data and is described nicely in [14]. If some or all of the random variables are continuous, then under certain assumptions again simple, standard methods exist. For example, under one set of assumptions linear regression can be used, and under another naive Bayes can be used. In fact, the work by Srinivasan and Camacho [35] on predicting levels of mutagenicity and the work by Craven and colleagues [4, 3] on information extraction can be seen as special cases of this proposed approach, employing linear regression and naive Bayes, respectively.

While the approach just outlined appears promising, of course it is not the only possible approach and may not turn out to be the best. More generally,

ILP and Bayes Net learning are largely orthogonal. The former handles relational domains well, while the latter handles probabilities well. And both Bayes Nets and ILP have been applied successfully to a variety of tasks. Therefore, it is reasonable to hypothesize the existence and utility of a representation and learning algorithms that effectively capture the advantages of both Bayes net learning and ILP. The space of such representations and algorithms is large, so combining Bayes Net learning and ILP is an area of research that is not only promising but also wide open for further work.

3.2 Stochastic Search

Most ILP algorithms search a lattice of clauses ordered by subsumption. They seek a clause that maximizes some function of the size of the clause and coverage of the clause, i.e. the numbers of positive and negative examples entailed by the clause together with the background theory. Depending upon how they search this lattice, these ILP algorithms are classified as either bottom-up (based on least general generalization) or top-down (based on refinement). Algorithms are further classified by whether they perform a greedy search, beam search, admissible search, etc. In almost all existing algorithms these searches are deterministic. But for other challenging logic/AI tasks outside ILP, stochastic searches have consistently outperformed deterministic searches. This observation has been repeated for a wide variety of tasks, beginning with the 1992 work of Kautz, Selman, Levesque, Mitchell, and others on satisfiability using algorithms such as GSAT and WSAT (WalkSAT) [34, 33]. Consequently, a promising research direction within ILP is the use of stochastic search rather than deterministic search to examine the lattice of clauses. A start has been made in stochastic search for ILP and this section describes that work. Nevertheless many issues remain unexamined, and I will mention some of the most important of these at the end of this section.

ILP algorithms face not one but two difficult search problems. In addition to the search of the lattice of clauses, already described, simply testing the coverage of a clause involves repeated searches for proofs—"if I assume this clause is true, does a proof exist for that example?" The earliest work on stochastic search in ILP (to my knowledge) actually addressed this latter search problem. Sebag and Rouveirol [32] employed stochastic matching, or theorem proving, and obtained efficiency improvements over Progol in the prediction of mutagenicity, without sacrificing predictive accuracy or comprehensibility. More recently, Botta, Giordana, Saitta, and Sebag have pursued this approach further, continuing to show the benefits of replacing deterministic matching with stochastic matching [11, 2].

But at the center of ILP is the search of the clause lattice, and surprisingly until now the only stochastic search algorithms that have been tested have been genetic algorithms. Within ILP these have not yet been shown to significantly outperform deterministic search algorithms. I say it is surprising that only GAs have been attempted because for other logical tasks such as satisfiability and planning almost every other approach outperforms GAs, including simulated annealing, hill-climbing with random restarts and sideways moves (e.g. GSAT),

and directed random walks (e.g. WSAT) [33]. Therefore, a natural direction for ILP research is to use these alternative forms of stochastic search to examine the lattice of clauses. The remainder of this section discusses some of the issues involved in this research direction, based on my initial foray in this direction with Ashwin Srinivasan that includes testing variants of GSAT and WSAT tailored to ILP.

The GSAT algorithm was designed for testing the satisfiability of Boolean CNF formulas. GSAT randomly draws a truth assignment over the n propositional variables in the formula and then repeatedly modifies the current assignment by flipping a variable. At each step all possible flips are tested, and the flip that yields the largest number of satisfied clauses is selected. It may be the case that every possible flip yields a score no better (in fact, possibly even worse) than the present assignment. In such a case a flip is still chosen and is called a "sideways move" (or "downward move" if strictly worse). Such moves turn out to be quite important in GSAT's performance. If GSAT finds an assignment that satisfies the CNF formula, it halts and returns the satisfying assignment. Otherwise, it continues to flip variables until it reaches some pre-set maximum number of flips. It then repeats the process by drawing a new random truth assignment. The overall process is repeated until a satisfying assignment is found or a pre-set maximum number of iterations is reached.

Our ILP variant of this algorithm draws a random clause rather than a random truth assignment. Flips involve adding or deleting literals in this clause. Applying the GSAT methodology to ILP in this manner raises several important points. First, in GSAT scoring a given truth assignment is very fast. In contrast, scoring a clause can be much more time consuming because it involves repeated theorem proving. Therefore, it might be beneficial to combine the "ILP-GSAT" algorithm with the type of stochastic theorem proving mentioned above. Second, the number of literals that can be built from a language often is infinite, so we cannot test all possible additions of a literal. Our approach has been to base any given iteration of the algorithm on a "bottom clause" built from a "seed example," based on the manner in which the ILP system PROGOL [26] constrains its search space. But there might be other alternatives for constraining the set of possible literals to be added at any step. Or it might be preferable to consider changing literals rather than only adding or deleting them. Hence there are many alternative GSAT-like algorithms that might be built and tested.

Based on our construction of GSAT-like ILP algorithms, one can imagine analogous WSAT-like and simulated annealing ILP algorithms. Consider WSAT in particular. On every flip, with probability p (user-specified) WSAT makes an randomly-selected efficacious flip instead of a GSAT flip. An efficacious flip is a flip that satisfies some previously-unsatisfied clause in the CNF formula, even if the flip is not the highest-scoring flip as required by GSAT. WSAT outperforms GSAT for many satisfiability tasks because the random flips make it less likely to get trapped in local optima. It will be interesting to see if the benefit of WSAT over GSAT for satisfiability carries over to ILP. The same issues mentioned above for ILP- GSAT also apply to ILP-WSAT.

It is too early in the work to present concrete conclusions regarding stochastic ILP. Rather the goal of this section has been to point to a promising direction and discuss the space of design alternatives to be explored. Researchers with experience in stochastic search for constraint satisfaction and other logic/AI search tasks will almost certainly have additional insights that will be vital to the exploration of stochastic search for ILP.

3.3 Special-Purpose Reasoning Mechanisms

One of the well-known success stories of computational logic is constraint logic programming. And one of the reasons for this success is the ability to integrate logic and special purpose reasoners or constraint solvers. Many ILP applications could benefit from the incorporation of special-purpose reasoning mechanisms. Indeed, the approach advocated in Section 3.1 to incorporating probabilities in ILP can be thought of as invoking special purpose reasoners to construct constraints in the form of Bayes Net fragments. The work by Srinivasan and Camacho mentioned there uses linear regression to construct a constraint, while the work by Craven and Slattery uses naive Bayes techniques to construct a constraint. The point that is crucial to notice is that ILP requires a "constraint constructor," such as linear regression, in addition to the constraint solver required during deduction. Let's now turn to consideration of tasks where other types of constraint generators might be useful.

Consider the general area of knowledge discovery from databases. Suppose we take the standard logical interpretation of a database, where each relation is a predicate, and each tuple in the relation is a ground atomic formula built from that predicate. Dzeroski and Lavrac show how ordinary ILP techniques are very naturally suited to this task, if we have an "ordinary" relational database. But now suppose the database contains some form of complex objects, such as images. Simple logical similarities may not capture the important common features across a set of images. Instead, special-purpose image processing techniques may be required, such as those described by Leung and colleagues [20, 19]. In addition to simple images, special-purpose constraint constructors might be required when applying ILP to movie (e.g. MPEG) or audio (e.g. MIDI) data, or other data forms that are becoming ever more commonplace with the growth of multimedia. For example, a fan of the Bach, Mozart, Brian Wilson, and Elton John would love to be able to enter her/his favorite pieces, have ILP with a constraint generator build rules to describe these favorites, and have the rules suggest other pieces or composers s/he should access. As multimedia data becomes more commonplace, ILP can remain applicable only if it is able to incorporate special-purpose constraint generators.

Alan Frisch and I have shown that the ordinary subsumption ordering over formulas scales up quite naturally to incorporate constraints [10]. Nevertheless, that work does not address some of the hardest issues, such as how to ensure the efficiency of inductive learning systems based on this ordering and how to design the right types of constraint generators. These questions require much further research involving real-world applications such as multimedia databases.

One final point about special purpose reasoners in ILP is worth making. Constructing a constraint may be thought of as inventing a predicate. Predicate invention within ILP has a long history [28, 39, 40, 25]. General techniques for predicate invention encounter the problem that the space of "inventable" predicates is unconstrained, and hence allowing predicate invention is roughly equivalent to removing all bias from inductive learning. While removing bias may sound at first to be a good idea, inductive learning in fact requires bias [23, 24]. Special purpose techniques for constraint construction appear to make it possible to perform predicate invention in way that is limited enough to be effective [35, 3].

3.4 Interaction with Human Experts

To discover new knowledge from data in fields such as telecommunications, molecular biology, or pharmaceuticals, it would be beneficial if a machine learning system and a human expert could act as a team, taking advantage of the computer's speed and the expert's knowledge and skills. ILP systems have three properties that make them natural candidates for collaborators with humans in knowledge discovery:

Declarative Background Knowledge ILP systems can make use of declarative background knowledge about a domain in order to construct hypotheses. Thus a collaboration can begin with a domain expert providing the learning system with general knowledge that might be useful in the construction of hypotheses. Most ILP systems also permit the expert to define the hypothesis space using additional background knowledge, in the form of a *declarative bias*.

Natural descriptions of structured examples Feature-based learning systems require the user to begin by creating features to describe the examples. Because many knowledge discovery tasks involve complex structured examples, such as molecules, users are forced to choose only composite features such as molecular weight—thereby losing information—or to invest substantial effort in building features that can capture structure (see [36] for a discussion in the context of molecules). ILP systems allow a structured example to be described naturally in terms of the objects that compose it, together with relations between those objects. The 2-dimensional structure of a molecule can be represented directly using its atoms as the objects and bonds as the relations; 3-dimensional structure can be captured by adding distance relations.

Human-Comprehensible Output ILP systems share with propositional–logic learners the ability to present a user with declarative, comprehensible rules as output. Some ILP systems can return rules in English along with visual aids. For example, the pharmacophore description and corresponding figures in Section 2 were generated automatically by PROGOL.

Despite the useful properties just outlined, ILP systems—like other machine learning systems—have a number of shortcomings as collaborators with humans

in knowledge discovery. One shortcoming is that most ILP systems return a single theory based on heuristics, thus casting away many clauses that might be interesting to a domain expert. But the only currently existing alternative is the version space approach, which has unpalatable properties that include inefficiency, poor noise tolerance, and a propensity to overwhelm users with too large a space of possible hypotheses. Second, ILP systems cannot respond to a human expert's questions in the way a human collaborator would. They operate in simple batch mode, taking a data set as input, and returning a hypothesis on a take-it-or-leave-it basis. Third, ILP systems do not question the input data in the way a human collaborator would, spotting surprising (and hence possibly erroneous) data points and raising questions about them. Some ILP systems will flag mutually inconsistent data points but to my knowledge none goes beyond this. Fourth, while a human expert can provide knowledge-rich forms of hypothesis justification, for example relating a new hypothesis to existing beliefs, ILP systems merely provide accuracy estimates as the sole justification.

To build upon ILP's strengths as a technology for human-computer collaboration in knowledge discovery, the above shortcomings should be addressed. ILP systems should be extended to display the following capabilities.

1. maintain and summarize alternative hypotheses that explain or describe the data, rather than providing a single answer based on a general-purpose heuristic;
2. propose to human experts practical sequences of experiments to refine or distinguish between competing hypotheses;
3. provide non-numerical justification for hypotheses, such as relating them to prior beliefs or illustrative examples (in addition to providing numerical accuracy estimates);
4. answer an expert's questions regarding hypotheses;
5. consult the expert regarding anomalies or surprises in the data.

Addressing such human-computer interface issues obviously requires a variety of logical and AI expertise. Thus contributions from other areas of computational logic, such as the study of logical agents, will be vital. While several projects have recently begun that investigate some of these issues,[6] developing collaborative systems is an ambitious goal with more than enough room for many more researchers. And undoubtedly other issues not mentioned here will become apparent as this work progresses.

3.5 Parallel Execution

While ILP has numerous advantages over other types of machine learning, including advantages mentioned at the start of the previous section, it has two

[6] Stephen Muggleton has a British EPSRC project on closed-loop learning, in which the human is omitted entirely. While this seems the reverse of a collaborative system, it raises similar issues, such as maintaining competing hypotheses and automatically proposing experiments. I am beginning a U.S. National Science Foundation project on collaborative systems with (not surprisingly) exactly the goals above.

particularly notable disadvantages—run time and space requirements. Fortunately for ILP, at the same time that larger applications are highlighting these disadvantages, parallel processing "on the cheap" is becoming widespread. Most notable is the widespread use of "Beowulf clusters" [1] and of "Condor pools" [21], arrangements that connect tens, hundreds, or even thousands of personal computers or workstations to permit parallel processing. Admittedly, parallel processing cannot change the order of the time or space complexity of an algorithm. But most ILP systems already use broad constraints, such as maximum clause size, to hold down exponential terms. Rather, the need is to beat back the large constants brought in by large real-world applications.

Yu Wang and David Skillicorn recently developed a parallel implementation of PROGOL under the Bulk Synchronous Parallel (BSP) model and claim superlinear speedup from this implementation [38]. Alan Wild worked with me at the University of Louisville to re-implement on a Beowulf cluster a top-down ILP search for pharmacophore discovery, and the result was a linear speedup [13]. The remainder of this section described how large-scale parallelism can be achieved very simply in a top-down complete search ILP algorithm. This was the approach taken in [13]. From this discussion, one can imagine more interesting approaches for other types of top-down searches such as greedy search.

The ideal in parallel processing is a decrease in processing time that is a linear function, with a slope near 1, of the number of processors used. (In some rare cases it is possible to achieve superlinear speed-up.) The barriers to achieving the ideal are (1) overhead in communication between processes and (2) competition for resources between processes. Therefore, a good parallel scheme is one where the processes are relatively independent of one another and hence require little communication or resource sharing. The key observation in the design of the parallel ILP scheme is that two competing hypotheses can be tested against the data completely independently of one another. Therefore the approach advocated here is to distribute the hypothesis space among different processors for testing against the data. These processors need not communicate with one another during testing, and they need not write to a shared memory space.

In more detail, for complete search a parallel ILP scheme can employ a master-worker design, where the master assigns different segments of the hypothesis space to workers that then test hypotheses against the data. Workers communicate back to the master all hypotheses achieving a pre-selected minimum valuation score (e.g. 95 % accuracy) on the data. As workers become free, the master continues to assign new segments of the space until the entire space has been explored. The only architectural requirements for this approach are (1) a mechanism for communication between the master and each worker and (2) read access for each worker to the data. Because data do not change during a run, this scheme can easily operate under either a shared memory or message passing architecture; in the latter, we incur a one-time overhead cost of initially communicating the data to each worker. The only remaining overhead, on either architecture, consists of the time spent by the master and time for master-worker communication. In "needle in a haystack" domains, which are the motivation

for complete search, one expects very few hypotheses to be communicated from workers to the master, so overhead for the communication of results will be low. If it also is possible for the master to rapidly segment the hypothesis space in such a way that the segments can be communicated to the workers succinctly, then overall overhead will be low and the ideal of linear speed-up can be realized.

4 Conclusions

ILP has attracted great interest within the machine learning and AI communities at large because of its logical foundations, its ability to utilize background knowledge and structured data representations, and its comprehensible results. But most of all, the interest has come from ILP's application successes. Nevertheless, ILP needs further advances to maintain this record of success, and these advances require further contributions from other areas of computational logic. System builders and parallel implementation experts are needed if the ILP systems of the next decade are to scale up to the next generation of data sets, such as those being produced by Affymetrix's (TM) gene expression microarrays and Celera's (TM) shotgun approach to DNA sequencing. Researchers on probability and logic are required if ILP is to avoid being supplanted by the next generation of extended Bayes Net learning systems. Experts on constraint satisfaction and constraint logic programming have the skills necessary to bring successful stochastic search techniques to ILP and to allow ILP techniques to extend to multimedia databases. The success of ILP in the next decade (notice I avoided the strong temptation to say "next millennium") depends on the kinds of interactions being fostered at Computational Logic 2000.

Acknowledgements

This work was supported in part by NSF grant 9987841 and by grants from the University of Wisconsin Graduate School and Medical School.

References

[1] D. Becker, T. Sterling, D. Savarese, E. Dorband, U. Ranawake, and C. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, pages 11–14, 1995.

[2] M. Botta, A. Giordana, L. Saiita, and M. Sebag. Relational learning: Hard problems and phase transitions. 2000.

[3] M. Craven and J. Kumlien. Constructing biological knowledge bases by extracting information from text sources. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, Heidelberg, Germany, 1999. AAAI Press.

[4] M. Craven and S. Slattery. Combining statistical and relational methods for learning in hypertext domains. In *Proceedings of the Eighth International Conference on Inductive Logic Programming (ILP-98)*, pages 38–52. Springer Verlag, 1998.

[5] J. Cussens. Loglinear models for first-order probabilistic reasoning. In *Proceedings of the 15th Conference on Uncertainty in Artificial Intelligence*. Stockholm, Sweden, 1999.
[6] S. Dzeroski. Inductive logic programming and knowledge discovery in databases. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. 1996.
[7] P. Finn, S. Muggleton, D. Page, and A. Srinivasan. Discovery of pharmacophores using Inductive Logic Programming. *Machine Learning*, 30:241–270, 1998.
[8] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*. Stockholm, Sweden, 1999.
[9] N. Friedman, D. Koller, and A. Pfeffer. Structured representation of complex stochastic systems. In *Proceedings of the 15th National Conference on Artificial Intelligence*. AAAI Press, 1999.
[10] A. M. Frisch and C. D. Page. Building theories into instantiation. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, 1995.
[11] A. Giordana and L. Saitta. Phase transitions in learning with fol languages. Technical Report 97, 1998.
[12] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24(2):109–111, 1987.
[13] J. Graham, D. Page, and A. Wild. Parallel inductive logic programming. In *Proceedings of the Systems, Man, and Cybernetics Conference (SMC-2000)*, page To appear. IEEE, 2000.
[14] D. Heckerman. A tutorial on learning with bayesian networks. Microsoft Technical Report MSR-TR-95-06, 1995.
[15] D. Heckerman, E. Horvitz, and B. Nathwani. Toward normative expert systems: Part i the pathfinder project. *Methods of Information in Medicine*, 31:90–105, 1992.
[16] R. King, S. Muggleton, R. Lewis, and M. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proceedings of the National Academy of Sciences*, 89(23):11322–11326, 1992.
[17] R. King, S. Muggleton, A. Srinivasan, and M. Sternberg. Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, 93:438–442, 1996.
[18] C. Lee. *A completeness theorem and a computer program for finding theorems derivable from given axioms*. PhD thesis, University of California, Berkeley, 1967.
[19] T. Leung, M. Burl, and P. Perona. Probabilistic affine invariants for recognition. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, 1998.
[20] T. Leung and J. Malik. Detecting, localizing and grouping repeated scene elements from images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page To appear, 2000.
[21] M. Litzkow, M. Livny, and M. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 104–111, 1988.
[22] J. Marcinkowski and L. Pacholski. Undecidability of the horn-clause implication problem. In *Proceedings of the 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 354–362. IEEE, 1992.

[23] T.M. Mitchell. The need for biases in learning generalizations. Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, 1980.
[24] T.M. Mitchell. Generalisation as search. *Artificial Intelligence*, 18:203–226, 1982.
[25] S. Muggleton. Predicate invention and utilization. *Journal of Experimental and Theoretical Artificial Intelligence*, 6(1):127–130, 1994.
[26] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
[27] S. Muggleton. Learning stochastic logic programs. In *Proceedings of the AAAI2000 Workshop on Learning Statistical Models from Relational Data*. AAAI, 2000.
[28] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Kaufmann, 1988.
[29] S. Muggleton, R. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.
[30] L. Ngo and P. Haddawy. Probabilistic logic programming and bayesian networks. *Algorithms, Concurrency, and Knowledge: LNCS 1023*, pages 286–300, 1995.
[31] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171:147–177, 1997.
[32] M. Sebag and C. Rouveirol. Tractable induction and classification in fol. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 888–892. Nagoya, Japan, 1997.
[33] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. AAAI Press, 1994.
[34] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446. AAAI Press, 1992.
[35] A. Srinivasan and R.C. Camacho. Numerical reasoning with an ILP system capable of lazy evaluation and customised search. *Journal of Logic Programming*, 40:185–214, 1999.
[36] A. Srinivasan, S. Muggleton, R. King, and M. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85(1,2):277–299, 1996.
[37] M. Turcotte, S. Muggleton, and M. Sternberg. Application of inductive logic programming to discover rules governing the three-dimensional topology of protein structures. In *Proceedings of the Eighth International Conference on Inductive Logic Programming (ILP-98)*, pages 53–64. Springer Verlag, 1998.
[38] Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. http://www.cs.queensu.ca/home/skill/papers.html#datamining, 2000.
[39] R. Wirth and P. O'Rorke. Constraints on predicate invention. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 457–461. Kaufmann, 1991.
[40] J. Zelle and R. Mooney. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 817–822, San Mateo, CA, 1993. Morgan Kaufmann.

A New Algorithm for Learning Range Restricted Horn Expressions*
(Extended Abstract)

Marta Arias and Roni Khardon

Division of Informatics, University of Edinburgh
The King's Buildings, Edinburgh EH9 3JZ, Scotland
{marta,roni}@dcs.ed.ac.uk

Abstract. A learning algorithm for the class of *range restricted Horn expressions* is presented and proved correct. The algorithm works within the framework of *learning from entailment*, where the goal is to exactly identify some pre-fixed and unknown expression by making questions to *membership* and *equivalence* oracles. This class has been shown to be learnable in previous work. The main contribution of this paper is in presenting a more direct algorithm for the problem which yields an improvement in terms of the number of queries made to the oracles. The algorithm is also adapted to the class of Horn expressions with inequalities on all syntactically distinct terms where a significant improvement in the number of queries is obtained.

1 Introduction

This paper considers the problem of learning an unknown first order expression[1] T from examples of clauses that T entails or does not entail. This type of learning framework is known as *learning from entailment*. [FP93] formalised learning from entailment using equivalence queries and membership queries and showed the learnability of propositional Horn expressions. Generalising this result to the first order setting is of clear interest. Learning first order Horn expressions has become a fundamental problem in *Inductive Logic Programming* [MR94]. Theoretical results have shown that learning from examples only is feasible for very restricted classes [Coh95a] and that, in fact, learnability becomes intractable when slightly more general classes are considered [Coh95b]. To tackle this problem, learners have been equipped with the ability to ask questions. It is the case that with this ability larger classes can be learned. In this paper, the questions that the learner is allowed to ask are *membership* and *equivalence* queries. While our work is purely theoretical, there are systems that are able to learn using equivalence and membership queries (MIS [Sha83], CLINT [RB92], for example). These ideas have also been used in systems that learn from examples only [Kha00].

* This work was partly supported by EPSRC Grant GR/M21409.

[1] The unknown expression to be identified is commonly referred to as *target* expression.

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 21–39, 2000.

A learning algorithm for the class of *range restricted Horn expressions* is presented. The main property of this class is that all the terms in the conclusion of a clause appear in the antecedent of the clause, possibly as subterms of more complex terms. This work is based on previous results on learnability of *function free Horn expressions* and *range restricted Horn expressions*. The problem of learning *range restricted Horn expressions* was solved in [Kha99b] by reducing it to the problem of learning *function free Horn expressions*, solved in [Kha99a]. The algorithm presented here has been obtained by retracing this reduction and using the resulting algorithm as a starting point. However, it has been significantly modified and improved. The algorithm in [Kha99a, Kha99b] uses two main procedures. The first, given a counterexample clause, minimises the clause while maintaining it as a counterexample. The minimisation procedure used here is stronger, resulting in a clause which includes a syntactic variant of a target clause as a subset. The second procedure combines two examples producing a new clause that may be a better approximation for the target. While the algorithm in [Kha99a, Kha99b] uses direct products of models we use an operation based on the *lgg* (least general generalisation [Plo70]). The use of *lgg* seems a more natural and intuitive technique to use for learning from entailment, and it has been used before, both in theoretical and applied work [Ari97, RT98, RS98, MF92].

We extend our results to the class of *fully inequated range restricted Horn expressions*. The main property of this class is that it does not allow unification of its terms. To avoid unification, every clause in this class includes in its antecedent a series of inequalities between all its terms. With a minor modification to the learning algorithm, we are able to show learnability of the class of fully inequated range restricted Horn expressions. The more restricted nature of this class allows for better bounds to be derived.

The rest of the paper is organised as follows. Section 2 gives some preliminary definitions. The learning algorithm is presented in Section 3 and proved correct in Section 4. The results are extended to the class of fully inequated range restricted Horn expressions in Section 5. Finally, Section 6 compares the results obtained in this paper with previous results and includes some concluding remarks.

2 Preliminaries

We consider a subset of the class of universally quantified expressions in first order logic. Definitions of first order languages can be found in standard texts, e.g. [Llo87]. We assume familiarity with notions such as *term*, *atom*, *literal*, *Horn clause* in the part of syntax and *interpretation*, *truth value*, *satisfiability* and *logical implication* in the part of semantics.

A *Range Restricted Horn clause* is a definite Horn clause in which every term appearing in its consequent also appears in its antecedent, possibly as a subterm of another term. A *Range Restricted Horn Expression* is a conjunction of Range Restricted Horn clauses.

A *multi-clause* is a pair of the form $[s, c]$, where both s and c are sets of literals such that $s \cap c = \emptyset$; s is the *antecedent* of the multi-clause and c is the *consequent*. Both are interpreted as the conjunction of the literals they contain. Therefore, the multi-clause $[s, c]$ is interpreted as the logical expression $\bigwedge_{b \in c} s \rightarrow b$. An ordinary clause $C = s_c \rightarrow b_c$ corresponds to the multi-clause $[s_c, \{b_c\}]$.

We say that a logical expression T *implies* a multi-clause $[s, c]$ if it implies all of its single clause components. That is, $T \models [s, c]$ iff $T \models \bigwedge_{b \in c} s \rightarrow b$.

A multi-clause $[s, c]$ is *correct* w.r.t an expression T iff $T \models [s, c]$. A multi-clause $[s, c]$ is *exhaustive* w.r.t T if every literal $b \notin s$ such that $T \models s \rightarrow b$ is included in c. A multi-clause is *full* w.r.t T if it is correct and exhaustive w.r.t. T.

The *size* of a term is the number of occurrences of variables plus twice the number of occurrences of function symbols (including constants). The *size* of an atom is the sum of the sizes of the (top-level) terms it contains plus 1. Finally, the *size* of a multi-clause $[s, c]$ is the sum of sizes of atoms in s.

Let s_1, s_2 be any two sets of literals. We say that s_1 *subsumes* s_2 (denoted $s_1 \preceq s_2$) if and only if there exists a substitution θ such that $s_1 \cdot \theta \subseteq s_2$. We also say that s_1 is a *generalisation* of s_2.

Let s be any set of literals. Then $ineq(s)$ is the set of all inequalities between terms appearing in s. As an example, let s be the set $\{p(x, y), q(f(y))\}$ with terms $\{x, y, f(y)\}$. Then $ineq(s) = \{x \neq y, x \neq f(y), y \neq f(y)\}$ also written as $(x \neq y \neq f(y))$ for short.

Least General Generalisation. The algorithm proposed uses the *least general generalisation* or lgg operation [Plo70]. This operation computes a generalisation of two sets of literals. It works as follows.

The lgg of two terms $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ is defined as the term $f(lgg(s_1, t_1), ..., lgg(s_n, t_n))$ if $f = g$ and $n = m$. Otherwise, it is a new variable x, where x stands for the lgg of that pair of terms throughout the computation of the lgg of the set of literals. This information is kept in what we call the lgg table. The lgg of two *compatible* atoms $p(s_1, ..., s_n)$ and $p(t_1, ..., t_n)$ is $p(lgg(s_1, t_1), ..., lgg(s_n, t_n))$. The lgg is only defined for compatible atoms, that is, atoms with the same predicate symbol and arity. The lgg of two *compatible* positive literals l_1 and l_2 is the lgg of the underlying atoms. The lgg of two *compatible* negative literals l_1 and l_2 is the negation of the lgg of the underlying atoms. Two literals are compatible if they share predicate symbol, arity and sign. The lgg of two sets of literals s_1 and s_2 is the set $\{lgg(l_1, l_2) \mid (l_1, l_2)$ are two compatible literals of s_1 and $s_2\}$.

Example 1. Let $s_1 = \{p(a, f(b)), p(g(a, x), c), q(a)\}$ and $s_2 = \{p(z, f(2)), q(z)\}$ with $lgg(s_1, s_2) = \{p(X, f(Y)), p(Z, V), q(X)\}$. The lgg table produced during the computation of $lgg(s_1, s_2)$ is

`[a - z => X]`	(from $p(\underline{a}, f(b))$ with $p(\underline{z}, f(2))$)
`[b - 2 => Y]`	(from $p(a, f(\underline{b}))$ with $p(z, f(\underline{2}))$)
`[f(b) - f(2) => f(Y)]`	(from $p(a, \underline{f(b)})$ with $p(z, \underline{f(2)})$)
`[g(a,x) - z => Z]`	(from $p(\underline{g(a,x)}, c)$ with $p(\underline{z}, f(2))$)
`[c - f(2) => V]`	(from $p(\underline{g(a,x)}, \underline{c})$ with $p(z, \underline{f(2)})$)

2.1 The Learning Model

We consider the model of *exact learning from entailment* [FP93]. In this model examples are clauses. Let T be the target expression, H any hypothesis presented by the learner and C any clause. An example C is positive for a target theory T if $T \models C$, otherwise it is negative. The learning algorithm can make two types of queries. An *Entailment Equivalence Query* ($EntEQ$) returns "Yes" if $H = T$ and otherwise it returns a clause C that is a counter example, i.e., $T \models C$ and $H \not\models C$ or vice versa. For an *Entailment Membership Query* ($EntMQ$), the learner presents a clause C and the oracle returns "Yes" if $T \models C$, and "No" otherwise. The aim of the learning algorithm is to exactly identify the target expression T by making queries to the equivalence and membership oracles.

2.2 Transforming the Target Expression

In this section we describe the transformation $U(T)$ performed on any target expression T. This transformation is never computed by the learning algorithm; it is only used in the analysis of the proof of correctness. Related work in [SEMF98] also uses inequalities in clauses, although the learning algorithm and approach are completely different.

The idea is to create from every clause C in T the set of clauses $U(C)$. Every clause in $U(C)$ corresponds to the original clause C with its terms unified in a unique way, different from every other clause in $U(C)$. All possible unifications of terms of C are covered by one of the clauses in $U(C)$. The clauses in $U(C)$ will only be satisfied if the terms are unified in exactly that way. To achieve this, a series of appropriate inequalities are prepended to every transformed clause's antecedent. This process is described in Figure 1. It uses the *most general unifier* operation or *mgu*. Details about the *mgu* can be found in [Llo87].

We construct $U(T)$ from T by considering every clause separately. For a clause C in T with set of terms $\mathcal{T}$, we generate a set of clauses $U(C)$. To do that, consider all partitions of the terms in $\mathcal{T}$; each such partition, say π, can generate a clause of $U(C)$, denoted $U_\pi(C)$. Therefore, $U(T) = \bigwedge_{C \in T} U(C)$ and $U(C) = \bigwedge_{\pi \in Partitions(\mathcal{T})} U_\pi(C)$. The clause $U_\pi(C)$ is computed as follows. Taking one class at a time, compute its *mgu* if possible. If there is no *mgu*, discard that partition. Otherwise, apply the unifying substitution to the rest of elements in classes not handled yet, and continue with the following class. If the representatives[2] of any two distinct classes happen to be equal, then discard that partition as well. This is because the inequality between the representatives of

[2] We call the representative of a class any of its elements applied to the *mgu*.

1. *Set $U(T)$ to be the empty expression (T is the expression to be transformed).*
2. *For every clause $C = s_c \to b_c$ in T and for every partition π of the set of terms (and subterms) appearing in C do*
   - *Let the partition π be $\{\pi_1, \pi_2, ..., \pi_l\}$. Set σ_0 to $\emptyset$.*
   - *For $i = 1$ to l do*
     - *If $\pi_i \cdot \sigma_{i-1}$ is unifiable, then $\theta_i = mgu(\pi_i \cdot \sigma_{i-1})$ and $\sigma_i = \sigma_{i-1} \cdot \theta_i$.*
     - *Otherwise, discard the partition.*
   - *If there are two classes π_i and π_j $(i \neq j)$ such that $\pi_i \cdot \sigma_l = \pi_j \cdot \sigma_l$, then discard the partition.*
   - *Otherwise, set $U_\pi(C) = ineq(s_c \cdot \sigma_l), s_c \cdot \sigma_l \to b_c \cdot \sigma_l$ and $U(T) = U(T) \wedge U_\pi(C)$.*
3. *Return $U(T)$.*

Fig. 1. The transformation algorithm

those two classes will never be satisfied (they are equal!), and the resulting clause is superfluous. When all classes have been unified, we proceed to translate the clause C. All (top-level) terms appearing in C are substituted by the *mgu* found for the class they appear in, and the inequalities are included in the antecedent. This gives the transformed clause $U_\pi(C)$.

Example 2. Let the clause to be transformed be $C = p(f(x), f(y), g(z)) \to q(x, y, z)$. The terms appearing in C are $\{x, y, z, f(x), f(y), g(z)\}$. We consider some possible partitions:

– When $\pi = \{x, y\}, \{z\}, \{f(x), f(y)\}, \{g(z)\}$.

Stage	*mgu*	θ	σ	*Partitions Left*
0			$\emptyset$	$\{x, y\}, \{z\}, \{f(x), f(y)\}, \{g(z)\}$
1	$\{x, y\}$	$\{y \mapsto x\}$	$\{y \mapsto x\}$	$\{z\}, \{f(x), f(x)\}, \{g(z)\}$
2	$\{z\}$	$\emptyset$	$\{y \mapsto x\}$	$\{f(x), f(x)\}, \{g(z)\}$
3	$\{f(x), f(x)\}$	$\emptyset$	$\{y \mapsto x\}$	$\{g(z)\}$
4	$\{g(z)\}$	$\emptyset$	$\{y \mapsto x\}$	

$C \cdot \sigma_4 = p(f(x), f(x), g(z)) \to q(x, x, z)$ and
$U_\pi(C) = (x \neq z \neq f(x) \neq g(z)), p(f(x), f(x), g(z)) \to q(x, x, z)$.

– When $\pi = \{x, y, z\}, \{f(x), g(z)\}, \{f(y)\}$.

Stage	*mgu*	θ	σ	*Partitions Left*
0			$\emptyset$	$\{x, y, z\}, \{f(x), g(z)\}, \{f(y)\}$
1	$\{x, y, z\}$	$\{y, z \mapsto x\}$	$\{y, z \mapsto x\}$	$\{f(x), g(x)\}, \{f(x)\}$
2	$\{f(x), g(x)\}$	No *mgu*		PARTITION DISCARDED

If the target expression T has m clauses, then the number of clauses in the transformation $U(T)$ is bounded by mt^t, with t being the maximum number of distinct terms appearing in one clause of T (the number of partitions of a set with t elements is bounded by t^t). Notice however that there are many partitions

that will be discarded by the process, for example all those partitions containing some class with two functional terms with different top-level function symbol, or partitions containing some class with two terms such that one is a subterm of the other. Therefore, the number of clauses in the transformation will be in practice much smaller than mt^t.

The transformation $U(T)$ of a range restricted expression T is also range restricted. It can be also proved that $T \models U(T)$, since every clause in $U(T)$ is subsumed by some clause in T. As a consequence, $U(T) \models C$ implies $T \models C$ and hence $U(T) \models [s, c]$ implies $T \models [s, c]$.

3 The Algorithm

The algorithm keeps a sequence S of representative counterexamples. The hypothesis H is generated from this sequence, and the main task of the algorithm is to *refine* the counterexamples in S in order to get a more accurate hypothesis in each iteration of the main loop, line 2, until hypothesis and target expression coincide.

There are two basic operations on counterexamples that need to be explained in detail. These are *minimisation* (line 2b), that takes a counterexample as given by the equivalence oracle and produces a positive, full counterexample; and *pairing* (line 2c), that takes two counterexamples and generates a series of candidate counterexamples. The counterexamples obtained by combination of previous ones (by *pairing* them) are the candidates to refine the sequence S. These operations are carefully explained in the following sections 3.1 and 3.2.

The algorithm uses the procedure rhs. The first version of rhs has 1 input parameter only. Given a set of literals s, $rhs(s)$ computes the set of all literals not in s implied by s w.r.t. the target expression. That is, $rhs(s) = \{b \notin s \mid EntMQ(s \rightarrow b) = Yes\}$.

The second version of rhs has 2 input parameters. Given the sets s and c, $rhs(s, c)$ outputs those literals in c that are implied by s w.r.t. the target expression. That is, $rhs(s, c) = \{b \in c \mid b \notin s \text{ and } EntMQ(s \rightarrow b) = Yes\}$. Notice that in both cases the literals b considered are literals containing terms that appear in s only. Both versions ask membership queries to find out which of the possible consequents are correct. The resulting sets are in both cases finite since the target expression is range restricted.

3.1 Minimising the Counterexample

The minimisation procedure has to transform a counterexample clause x as generated by the equivalence query oracle into a multi-clause counterexample $[s_x, c_x]$ ready to be handled by the learning algorithm. This is done by removing literals and generalising terms.

The minimisation procedure constructs first a full multi-clause that will be refined in the following steps. To do this, all literals implied by $antecedent(x)$ and the clauses in the hypothesis will be included in the first version of the new

1. *Set S to be the empty sequence and H to be the empty hypothesis.*
2. *Repeat until $EntEQ(H)$ returns "Yes":*
   (a) *Let x be the (positive) counterexample received ($T \models x$ and $H \not\models x$).*
   (b) *Minimise counterexample x - use calls to $EntMQ$.*
   Let $[s_x, c_x]$ be the minimised counterexample produced.
   (c) *Find the first $[s_i, c_i] \in S$ such that there is a basic pairing $[s, c]$ of terms of $[s_i, c_i]$ and $[s_x, c_x]$ satisfying:*
      i. $size(s) \lneq size(s_i)$
      ii. $rhs(s, c) \neq \emptyset$
   (d) *If such an $[s_i, c_i]$ is found then replace it by the multi-clause $[s, rhs(s, c)]$.*
   (e) *Otherwise, append $[s_x, c_x]$ to S.*
   (f) *Set H to be $\bigwedge_{[s,c]\in S} \{s \rightarrow b \mid b \in c\}$.*
3. *Return H*

Fig. 2. The learning algorithm

1. *Let x be the counterexample obtained by the $EntEQ$ oracle.*
2. *Let s_x be the set of literals $\{b \mid H \models antecedent(x) \rightarrow b\}$ and set c_x to $rhs(s_x)$.*
3. *Repeat until no more changes are made*
   – *For every functional term t appearing in s_x, in decreasing order of size, do*
      • *Let $[s'_x, c'_x]$ be the multi-clause obtained from $[s_x, c_x]$ after substituting all occurrences of the term $f(\overline{t})$ by a new variable $x_{f(\overline{t})}$.*
      • *If $rhs(s'_x, c'_x) \neq \emptyset$, then set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$.*
4. *Repeat until no more changes are made*
   – *For every term t appearing in s_x, in increasing order of size, do*
      • *Let $[s'_x, c'_x]$ be the multi-clause obtained after removing from $[s_x, c_x]$ all those literals containing t.*
      • *If $rhs(s'_x, c'_x) \neq \emptyset$, then set $[s_x, c_x]$ to $[s'_x, rhs(s'_x, c'_x)]$.*
5. *Return $[s_x, c_x]$.*

Fig. 3. The minimisation procedure

counterexample's antecedent: s_x (line 2). This can be done by forward chaining using the hypothesis' clauses, starting with the literals in $antecedent(x)$. Finally, the consequent of the first version of the new counterexample (c_x) will be constructed as $rhs(s_x)$.

Next, we enter the loop in which terms are generalised (line 3). We do this by considering every term that is not a variable (constants are also included), one at a time. The way to proceed is to substitute every occurrence of the term by a new variable, and then check whether the multi-clause is still positive. If so, the counterexample is updated to the new multi-clause obtained. The process finishes when there are no terms to be generalised in $[s_x, c_x]$. Note that if some term cannot be generalised, it will stay so during the computation of this loop, so that by keeping track of the failures, unnecessary computation time and queries can be saved.

Finally, we enter the loop in which literals are removed (line 4). We do this by considering one term at a time. We remove every literal containing that term in s_x and c_x and check if the multi-clause is still positive. If so, the counterexample is updated to the new multi-clause obtained. The process finishes when there are no terms to be dropped in $[s_x, c_x]$.

Example 3. Parentheses are omitted and the function f is unary. Let T be the single clause $p(fx) \rightarrow q(x)$. We start with counterexample $[p(fa), q(b) \rightarrow q(a)]$ as obtained after step 2 of the minimisation procedure.

$[s_x, c_x]$	*Stage*	$[s'_x, c'_x]$	$rhs(s'_x, c'_x)$	*To check*
	GEN			$\{fa, a, b\}$
$[p(fa), q(b) \rightarrow q(a)]$	$fa \mapsto X$	$[p(X), q(b) \rightarrow q(a)]$	$\emptyset$	$\{a, b\}$
$[p(fa), q(b) \rightarrow q(a)]$	$a \mapsto X$	$[p(fX), q(b) \rightarrow q(X)]$	$q(X)$	$\{b\}$
$[p(fX), q(b) \rightarrow q(X)]$	$b \mapsto Y$	$[p(fX), q(Y) \rightarrow q(X)]$	$q(X)$	$\{\}$
	DROP			$\{X, Y, fX\}$
$[p(fX), q(Y) \rightarrow q(X)]$	X	$[q(Y) \rightarrow \emptyset]$	$\emptyset$	$\{Y, fX\}$
$[p(fX), q(Y) \rightarrow q(X)]$	Y	$[p(fX) \rightarrow q(X)]$	$q(X)$	$\{fX\}$
$[p(fX) \rightarrow q(X)]$	fX	$[\emptyset \rightarrow q(X)]$	$\emptyset$	$\{\}$
$[p(fX) \rightarrow q(X)]$				

3.2 Pairings

A crucial process in the algorithm is how two counterexamples are combined into a new one, hopefully yielding a better approximation of some target clause. The operation proposed here uses pairings of clauses, based on the *lgg*.

We have two multi-clauses, $[s_x, c_x]$ and $[s_i, c_i]$ that need to be combined. To do so, we generate a series of matchings between the terms of s_x and s_i, and any of these matchings will produce the candidate to refine the sequence S.

Matchings. A matching is a set whose elements are pairs of terms $t_x - t_i$, where $t_x \in s_x$ and $t_i \in s_i$. If s_x contains less terms than s_i, then there should be an entry in the matching for every term in s_x. Otherwise, there should be an entry for every term in s_i. That is, the number of entries in the matching equals the minimum of the number of terms in s_x and s_i. We only use 1-1 matchings, i.e., once a term has been included in the matching it cannot appear in any other entry of the matching. Usually, we denote a matching by the Greek letter σ.

Example 4. Let $[s_x, c_x]$ be $[\{p(a, b)\}, \{q(a)\}]$ with terms $\{a, b\}$. Let $[s_i, c_i]$ be $[\{p(f(1), 2)\}, \{q(f(1))\}]$ with terms $\{1, 2, f(1)\}$. The possible matchings are:

$$\sigma_1 = \{a - 1, b - 2\} \quad \sigma_3 = \{a - 2, b - 1\} \quad \sigma_5 = \{a - f(1), b - 1\}$$
$$\sigma_2 = \{a - 1, b - f(1)\} \quad \sigma_4 = \{a - 2, b - f(1)\} \quad \sigma_6 = \{a - f(1), b - 2\}$$

An *extended matching* is an ordinary matching with an extra column added to every entry of the matching. This extra column contains the *lgg* of every pair in the matching. The *lgg*s are simultaneous, that is, they share the same table.

An extended matching σ is *legal* if every subterm of some term appearing as the *lgg* of some entry, also appears as the *lgg* of some other entry of σ. An ordinary matching is legal if its extension is.

Example 5. Let σ_1 be $\{a - c, f(a) - b, f(f(a)) - f(b), g(f(f(a))) - g(f(f(c)))\}$ and $\sigma_2 = \{a - c, f(a) - b, f(f(a)) - f(b)\}$. The matching σ_1 is not legal, since the term $f(X)$ is not present in its extension column and it is a subterm of $g(f(f(X)))$, which is present. The matching σ_2 is legal.

Extended σ_1

```
[a - c => X]
[f(a) - b => Y]
[f(f(a)) - f(b) => f(Y)]
[g(f(f(a))) - g(f(f(c))) => g(f(f(X)))]
```

Extended σ_2

```
[a - c => X]
[f(a) - b => Y]
[f(f(a)) - f(b) => f(Y)]
```

Our algorithm considers yet a more restricted type of matching. A *basic matching* σ is defined for two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$ such that the number of terms in s_x is less than or equal to the number of terms in s_i. It is a 1-1, legal matching such that if entry $f(t_1, ..., t_n) - g(r_1, ..., r_m) \in \sigma$, then $f = g$, $n = m$ and $t_i - r_i \in \sigma$ for all $i = 1, ..., n$. Notice this is not a symmetric operation, since $[s_x, c_x]$ is required to have less distinct terms than $[s_i, c_i]$.

To construct basic matchings given $[s_x, c_x]$ and $[s_i, c_i]$, consider all possible matchings between the *variables* in s_x and the *terms* in s_i only. Complete them by adding the functional terms in s_x that are not yet included in the basic matching in an upwards fashion, beginning with the more simple terms. For every term $f(t_1, ..., t_n)$ in s_x such that all $t_i - r_i$ (with $i = 1, ..., n$) appear already in the basic matching, add a new entry $f(t_1, ..., t_n) - f(r_1, ..., r_n)$. Notice this is not possible if $f(r_1, ..., r_n)$ does not appear in s_i or the term $f(r_1, ..., r_n)$ has already been used. In this case, we cannot complete the matching and it is discarded. Otherwise, we continue until all terms in s_x appear in the matching. By construction, constants in s_x must be matched to the same constants in s_i.

Example 6. Let s_x be $\{p(a, fx)\}$ containing the terms $\{a, x, fx\}$. Let s_i be $\{p(a, f1), p(a, 2)\}$ containing terms $\{a, 1, 2, f1\}$. No parentheses for functions are written. The basic matchings to consider are the following.

- Starting with `[x - a]`: cannot add `[a - a]`, therefore discarded.
- Starting with `[x - 1]`: can be completed with `[a - a]` and `[fx - f1]`.
- Starting with `[x - 2]`: cannot add `[fx - f2]`, therefore discarded.
- Starting with `[x - f1]`: cannot add `[fx - ff1]`, therefore discarded.

One of the key points of our algorithm lies in reducing the number of matchings needed to be checked by ruling out some of the candidate matchings that do not satisfy some restrictions imposed. By doing so we avoid testing too many pairings and hence avoid making unnecessary calls to the oracles. One of the restrictions has already been mentioned, it consists in considering basic pairings

only as opposed to considering every possible matching. This reduces the t^t possible distinct matchings to only t^k distinct *basic* pairings[3]. The other restriction on the candidate matching consists in the fact that every one of its entries must appear in the original lgg table.

Pairings. The input to a pairing consists of 2 multi-clauses together with a matching between the terms appearing in them. We say that the pairing is *induced* by the matching under consideration. A *basic* pairing is a pairing for which the inducing matching is basic.

The antecedent s of the pairing is computed as the lgg of s_x and s_i restricted to the matching inducing it. An atom is included in the pairing only if all its top-level terms appear as entries in the extended matching. This restriction is quite strong in the sense that, for example, if an atom $p(a)$ appears in both s_x and s_i then their lgg $p(a)$ will not be included unless the entry `[a - a => a]` appears in the matching. Therefore an entry in the matching is relevant only if it appears in the lgg table. We consider matchings with relevant entries only, i.e., matchings that are subsets of the lgg table.

To compute the consequent c of the pairing, the union of the sets $lgg_{|\sigma}(s_x, c_i)$, $lgg_{|\sigma}(c_x, s_i)$ and $lgg_{|\sigma}(c_x, c_i)$ is computed. Note that in the consequent all the possible lggs of pairs among $\{s_x, c_x, s_i, c_i\}$ are included except $lgg_{|\sigma}(s_x, s_i)$, that constitutes the antecedent. When computing any of the lggs, the same table is used. That is, the same pair of terms will be bound to the same expression in any of the four possible lggs that are computed in a pairing. To summarise:

$$[s, c] = [lgg_{|\sigma}(s_x, s_i), lgg_{|\sigma}(s_x, c_i) \cup lgg_{|\sigma}(c_x, s_i) \cup lgg_{|\sigma}(c_x, c_i)].$$

Example 7. Both examples have the same terms as in Example 6, so there is only one basic matching. Ex. 7.1 shows how to compute a pairing. Ex. 7.2 shows that a basic matching may be rejected if it does not agree with the lgg table.

	Example 7.1	*Example 7.2*
s_x	$\{p(a, fx)\}$	$\{p(a, fx)\}$
s_i	$\{p(a, f1), p(a, 2)\}$	$\{q(a, f1), p(a, 2)\}$
$lgg(s_x, s_i)$	$\{p(a, fX), p(a, Y)\}$	$\{p(a, Y)\}$

lgg table	`[a - a => a]`
	`[x - 1 => X]`
	`[fx - f1 => fX]`
	`[fx - 2 => Y]`
		`[a - a => a]`
		`[fx - 2 => Y]`

basic σ	`[a-a=>a]` `[x-1=>X]` `[fx-f1=>fX]`	`[a-a=>a]` `[x-1=>X]` `[fx-f1=>fX]`
$lgg_{\|\sigma}(s_x, s_i)$	$\{p(a, fX)\}$	PAIRING REJECTED

As the examples demonstrate, the requirement that the matchings are both basic and comply with the lgg table is quite strong. The more structure examples

[3] There are a maximum of t^k basic matchings between $[s_x, c_x]$ with k variables and $[s_i, c_i]$ with t terms, since we only combine *variables* of s_x with *terms* in s_i.

have, the greater the reduction in possible pairings and hence queries is, since that structure needs to be matched. While it is not possible to quantify this effect without introducing further parameters, we expect this to be a considerable improvement in practice.

4 Proof of Correctness

During the analysis, s will stand for the cardinality of P, the set of predicate symbols in the language; a for the maximal arity of the predicates in P; k for the maximum number of distinct variables in a clause of T; t for the maximum number of distinct terms in a clause of T; e_t for the maximum number of distinct terms in a counterexample; m for the number of clauses of the target expression T; m' for the number of clauses of the transformation of the target expression $U(T)$ as described in Section 2.2. Due to lack of space, some proofs have been reduced to simple sketches or even omitted. For a detailed account of the proof, see [AK00b]. Before starting with the proof, we give some definitions.

A multi-clause $[s, c]$ *covers* a clause $ineq(s_t), s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $s_t \cdot \theta \subseteq s$ and $ineq(s_t) \cdot \theta \subseteq ineq(s)$. Equivalently, we say that $ineq(s_t), s_t \rightarrow b_t$ *is covered* by $[s, c]$.

A multi-clause $[s, c]$ *captures* a clause $ineq(s_t), s_t \rightarrow b_t$ if there is a mapping θ from variables in s_t into terms in s such that $ineq(s_t), s_t \rightarrow b_t$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \in c$. Equivalently, we say that $ineq(s_t), s_t \rightarrow b_t$ *is captured by* $[s, c]$.

It is clear that if the algorithm stops, then the returned hypothesis is correct. Therefore the proof focuses on assuring that the algorithm finishes. To do so, a bound is established on the length of the sequence S. That is, only a finite number of counterexamples can be added to S and every refinement of an existing multi-clause reduces its size, and hence termination is guaranteed.

Lemma 1. *If $[s, c]$ is a positive example for a Horn expression T, then there is some clause $ineq(s_t), s_t \rightarrow b_t$ of $U(T)$ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t) \cdot \theta \subseteq ineq(s)$ and $b_t \cdot \theta \notin s$, where θ is some substitution mapping variables of s_t into terms of s. That is, $ineq(s_t), s_t \rightarrow b_t$ is covered by $[s, c]$ via θ and $b_t \cdot \theta \notin s$.*

Proof. Consider the interpretation I whose objects are the different terms appearing in s plus an additional special object $*$. Let D_I be the set of objects in I. Let σ be the mapping from terms in s into objects in I. The function mappings in I are defined following σ, or $*$ when not specified. We want I to falsify the multi-clause $[s, c]$. Therefore, the extension of I, say $ext(I)$, includes exactly those literals in s (with the corresponding new names for the terms), that is, $ext(I) = s \cdot \sigma$, where the top-level terms in s are substituted by the image in D_I given by σ.

It is easy to see that this I falsifies $[s, c]$, because $s \cap c = \emptyset$ by definition of multi-clause. Since $I \not\models [s, c]$ and $T \models [s, c]$, we can conclude that $I \not\models T$. That is, there is a clause $C = s_c \rightarrow b_c$ in T such that $I \not\models C$ and there is a substitution

θ' from variables in s_c into domain objects in I such that $s_c \cdot \theta' \subseteq ext(I)$ and $b_c \cdot \theta' \notin ext(I)$.

Complete the substitution θ' by adding all the remaining functional terms of C. The image that they are assigned to is their interpretation using the function mappings in I and the variable assignment θ'. When all terms have been included, consider the partition π induced by the completed θ', that is, two terms are included in the same class of the partition iff they are mapped to the same domain object by the completed θ'. Now, consider the clause $U_\pi(C)$. This clause is included in $U(T)$ because the classes are unifiable (the existence of $[s, c]$ is the proof for it) and therefore it is not rejected by the transformation procedure.

We claim that this clause $U_\pi(C)$ is the clause $ineq(s_t), s_t \rightarrow b_t$ mentioned in the lemma. Let $\hat{\theta}$ be the mgu used to obtain $U_\pi(C)$ from C with the partition π. That is, $U_\pi(C) = ineq(s_c \cdot \hat{\theta}), s_c \cdot \hat{\theta} \rightarrow b_c \cdot \hat{\theta}$. Let θ'' be the substitution such that $\theta' = \hat{\theta} \cdot \theta''$. The substitution θ'' exists since $\hat{\theta}$ is a mgu and by construction θ' is also a unifier for every class in the partition. The clause $U_\pi(C) = ineq(s_t), s_t \rightarrow b_t$ is falsified using the substitution θ'': $s_c \cdot \theta' = s_c \cdot \hat{\theta} \cdot \theta'' = s_t \cdot \theta'' \subseteq ext(I)$, and $b_c \cdot \theta' = b_c \cdot \hat{\theta} \cdot \theta'' = b_t \cdot \theta'' \notin ext(I)$.

Now we have to find a θ for which the three conditions stated in the lemma are satisfied. We define θ as $\theta'' \cdot \sigma^{-1}$. Notice σ is invertible since all the elements in its range are different. It can also be composed to θ'' since all elements in the range of θ'' are in D_I, and the domain of σ consists precisely of all objects in D_I. Notice also that $s = ext(I) \cdot \sigma^{-1}$, and this can be done since the object $*$ does not appear in $ext(I)$. It is left to show that:

– $s_t \cdot \theta \subseteq s$: $s_t \cdot \theta'' \subseteq ext(I)$ implies $s_t \cdot \theta = s_t \cdot \theta'' \cdot \sigma^{-1} \subseteq ext(I) \cdot \sigma^{-1} = s$.

– $ineq(s_t) \cdot \theta \subseteq ineq(s)$. Take any two different terms t, t' of s_t. The inequality $t \neq t' \in ineq(s_t)$, since we have assumed they are different. The terms $t \cdot \theta, t' \cdot \theta$ appear in s, since $s_t \cdot \theta \subseteq s$. In order to be included in $ineq(s)$ they need to be different terms. Hence, we only need to show that the terms $t \cdot \theta, t' \cdot \theta$ are different terms. By way of contradiction, suppose they are not, i.e. $t \cdot \theta = t' \cdot \theta$, so that $t \cdot \theta'' \cdot \sigma^{-1} = t' \cdot \theta'' \cdot \sigma^{-1}$. The substitution σ^{-1} maps different objects into different terms, hence t and t' were mapped into the same domain object of I by θ''. Or equivalently, the terms t_c, t'_c of s_c for which $t = t_c \cdot \hat{\theta}$ and $t' = t'_c \cdot \hat{\theta}$ were mapped into the same domain object. But then they fall into the same class of the partition, hence they have the same representative in s_t and $t = t_c \cdot \hat{\theta} = t'_c \cdot \hat{\theta} = t'$, which contradicts our assumption that t and t' are different.

– $b_t \cdot \theta \notin s$: $b_t \cdot \theta'' \notin ext(I)$ implies $b_t \cdot \theta = b_t \cdot \theta'' \cdot \sigma^{-1} \notin ext(I) \cdot \sigma^{-1} = s$. ■

Lemma 1 implies that every full multi-clause w.r.t. T, say $[s, c]$, captures some clause in $U(T)$. This is because $[s, c]$ is full and $b_t \cdot \theta \notin s$, and hence $b_t \cdot \theta \in c$. Also, $rhs(s, c)$ cannot be empty since the correct $b_t \cdot \theta \in c$ must survive.

A multi-clause $[s, c]$ is a positive counterexample for some target expression T and some hypothesis H if $T \models [s, c]$, $c \neq \emptyset$ and for all literals $b \in c$, $H \not\models s \rightarrow b$. Notice that the hypothesis H is always entailed by the target T and therefore the equivalence oracle can only give positive counterexamples. This is because all multi-clauses in the sequence S are correct at any time.

Let $[s_x, c_x]$ be any minimised counterexample. Then, the multi-clause $[s_x, c_x]$ is full w.r.t. T and it is a positive counterexample w.r.t. target T and hypothesis H. Both properties can be proved by induction on the number of times $[s_x, c_x]$ is updated during the minimisation process.

Lemma 2. *Let $[s_x, c_x]$ be a multi-clause as generated by the minimisation procedure. If $[s_x, c_x]$ captures some clause $ineq(s_t), s_t \rightarrow b_t$ of $U(T)$, then it must be via some substitution θ such that θ is a variable renaming, i.e., θ maps distinct variables of s_t into distinct variables of s_x only.*

Proof. $[s_x, c_x]$ is capturing $ineq(s_t), s_t \rightarrow b_t$, hence there must exist a substitution θ from variables in s_t into terms in s_x such that $s_t \cdot \theta \subseteq s_x$, $ineq(s_t) \cdot \theta \subseteq ineq(s_x)$ and $b_t \cdot \theta \in c_x$. We will show that θ must be a variable renaming.

By way of contradiction, suppose that θ maps some variable v of s_t into a functional term t of s_x (i.e. $v \cdot \theta = t$). Consider the generalisation of the term t in step 3 of the minimisation procedure. We will see that the term t should have been generalised and substituted by the new variable x_t, contradicting the fact that the variable v was mapped into a functional term.

Let $\theta_t = \{t \mapsto x_t\}$ and $[s'_x, c'_x] = [s_x \cdot \theta_t, c_x \cdot \theta_t]$. Consider the substitution $\theta \cdot \theta_t$. We will see that $[s'_x, c'_x]$ captures $ineq(s_t), s_t \rightarrow b_t$ via $\theta \cdot \theta_t$ and hence $rhs(s'_x, c'_x) \neq \emptyset$ and therefore t must be generalised to the variable x_t. To see this we need to show:

– $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$. By hypothesis $s_t \cdot \theta \subseteq s_x$ implies $s_t \cdot \theta \cdot \theta_t \subseteq s_x \cdot \theta_t = s'_x$.

– $ineq(s_t) \cdot \theta \cdot \theta_t \subseteq ineq(s'_x)$. Let t_1, t_2 two distinct terms of s_t. We have to show that $t_1 \cdot \theta \cdot \theta_t$ and $t_2 \cdot \theta \cdot \theta_t$ are two different terms of s'_x and therefore their inequality appears in $ineq(s'_x)$. It is easy to see that they are terms of s'_x since $s_t \cdot \theta \cdot \theta_t \subseteq s'_x$. To see that they are also different terms, notice first that $t_1 \cdot \theta$ and $t_2 \cdot \theta$ are different terms of s_x, since the clause $ineq(s_t), s_t \rightarrow b_t$ is captured by $[s_x, c_x]$. It is sufficient to show that if t'_1, t'_2 are any two distinct terms of s_x, then $t'_1 \cdot \theta_t$ and $t'_2 \cdot \theta_t$ are also distinct terms.

Notice the substitution θ_t maps the term t into a new variable x_t that does not appear in s_x. Consider the first position where t'_1 and t'_2 differ. Then, $t'_1 \cdot \theta_t$ and $t'_2 \cdot \theta_t$ will also differ in this same position, since at most one of the terms can contain t in that position. Therefore they also differ after applying θ_t.

– $b_t \cdot \theta \cdot \theta_t \in c'_x$. By hypothesis $b_t \cdot \theta \in c_x$ implies $b_t \cdot \theta \cdot \theta_t \in c_x \cdot \theta_t = c'_x$. ■

Another property satisfied by minimised multi-clauses is that the number of distinct terms appearing in a minimised multi-clause coincides with the number of distinct terms of any clause of $U(T)$ it captures. This is because the minimisation procedure detects and drops the superfluous term in s_x. Let t be the bound for the number of distinct terms in any clause of $U(T)$. Then, t bounds the number of distinct terms of any minimised multi-clause.

Let $[s_i, c_i]$ be any multi-clause covering a clause in $U(T)$. It can be shown that the number of distinct terms in s_i is greater or equal than the number of terms in the clause it covers. This happens because the substitution showing the covering of $[s_i, c_i]$ does not unify terms. We can conclude that if $[s_i, c_i]$ covers a clause captured by a minimised $[s_x, c_x]$, then s_x has no more terms than s_i.

It can be shown that if $[s_x, c_x]$ and $[s_i, c_i]$ are two full multi-clauses w.r.t. the target expression T, σ is a basic matching between the terms in s_x and s_i that is not rejected by the pairing procedure, and $[s, c]$ is the basic pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ, then the multi-clause $[s, rhs(s, c)]$ is also full w.r.t. T. This, together with the fact that every minimised counterexample $[s_x, c_x]$ is full w.r.t. T implies that every multi-clause in the sequence S is full w.r.t. T, since its elements are constructed by initially being a full multi-clause and subsequently pairing full multi-clauses.

Lemma 3. *Let S be the sequence $[[s_1, c_1], [s_2, c_2], ..., [s_k, c_k]]$. If a minimised counterexample $[s_x, c_x]$ is produced such that it captures some clause in $U(T)$ $ineq(s_t), s_t \rightarrow b_t$ covered by some $[s_i, c_i]$ of S, then some multi-clause $[s_j, c_j]$ will be replaced by a basic pairing of $[s_x, c_x]$ and $[s_j, c_j]$, where $j \leq i$.*

Proof (Sketch). We will show that if no element $[s_j, c_j]$ where $j < i$ is replaced, then the element $[s_i, c_i]$ will be replaced. We have to prove that there is a basic pairing $[s, c]$ of $[s_x, c_x]$ and $[s_i, c_i]$ with the following two properties: $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$.

We have assumed that there is some clause $ineq(s_t), s_t \rightarrow b_t \in U(T)$ captured by $[s_x, c_x]$ and covered by $[s_i, c_i]$. Let θ'_x be the substitution showing $ineq(s_t), s_t \rightarrow b_t$ being captured by $[s_x, c_x]$ and θ'_i the substitution showing $ineq(s_t), s_t \rightarrow b_t$ being covered by $[s_i, c_i]$. Thus the following holds: $s_t \cdot \theta'_x \subseteq s_x$; $ineq(s_t) \cdot \theta'_x \subseteq ineq(s_x)$; $b_t \cdot \theta'_x \in c_x$ and $s_t \cdot \theta'_i \subseteq s_i$; $ineq(s_t) \cdot \theta'_i \subseteq ineq(s_i)$.

We construct a matching σ that includes all entries `[`$t \cdot \theta'_x$ `-` $t \cdot \theta'_i$ `=>` $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$`]` such that t is a term appearing in s_t (only one entry for every distinct term of s_t).

Example 8. Let s_t be $\{p(g(c), x, f(y), z)\}$ containing 6 terms: c, $g(c)$, x, y, $f(y)$ and z. Let s_x be $\{p(g(c), x', f(y'), z), p(g(c), g(c), f(y'), c)\}$, containing 6 terms: c, $g(c)$, x', y', $f(y')$, z. Let s_i be $\{p(g(c), f(1), f(f(2)), z)\}$, containing 8 terms: c, $g(c)$, 1, $f(1)$, 2, $f(2)$, $f(f(2))$, z. The substitution θ'_x is $\{x \mapsto x', y \mapsto y', z \mapsto z\}$ and it is a variable renaming. The substitution θ'_i is $\{x \mapsto f(1), y \mapsto f(2), z \mapsto z\}$.

The $lgg(s_x, s_i)$ is $\{p(g(c), X, f(Y), z), p(g(c), Z, f(Y), V)\}$ and it produces the following lgg table.

```
[c - c => c]          [g(c) - g(c) => g(c)]     [x' - f(1) => X]
[y' - f(2) => Y]      [f(y') - f(f(2)) => f(Y)] [z - z => z]
[g(c) - f(1) => Z]    [c - z => V]
```

The $lgg_{|\sigma}(s_x, s_i)$ is $\{p(g(c), X, f(Y), z)\}$ and the extended matching σ is

```
    c ⇒ [c - c => c]                  g(c) ⇒ [g(c) - g(c) => g(c)])
    x ⇒ [x' - f(1) => X]                y ⇒ [y' - f(2) => Y]
f(y) ⇒ [f(y') - f(f(2)) => f(Y)]        z ⇒ [z - z => z]
```

The matching σ as described above is 1-1 and it is not discarded by the pairing procedure. Moreover, the number of entries in σ equals the minimum of the number of distinct terms in s_x and s_i. Let $[s, c]$ denote the pairing of $[s_x, c_x]$ and $[s_i, c_i]$ induced by σ.

We first claim that the matching σ is legal and basic. This can be shown by induction on the structure of the terms in s_t that induce every entry in σ. The induction hypothesis is the following. If t is a term in s_t, then the term $lgg(t \cdot \theta'_x, t \cdot \theta'_i)$ and all its subterms appear in the extension of some other entries of σ. The induction uses the fact that s_x contains a variant of s_t. Thus σ is considered by the algorithm.

Next, we argue that the conditions $rhs(s, c) \neq \emptyset$ and $size(s) \lneq size(s_i)$ hold. Let θ be the substitution that maps all variables in s_t to their corresponding expression assigned in the extension of σ. That is, θ maps any variable v of s_t to the term $lgg(v \cdot \theta'_x, v \cdot \theta'_i)$. In our example, $\theta = \{x \mapsto X, y \mapsto Y, z \mapsto z\}$. The proof of $rhs(s, c) \neq \emptyset$ consists in showing that $[s, c]$ captures $ineq(s_t), s_t \rightarrow b_t$ via θ.

The use of 1-1 matchings in pairings guarantees that the number of literals in the antecedent of a pairing never exceeds that of the multi-clauses originating it, since at most one copy of every atom in the original multi-clauses is included in the pairing. Thus, $size(s) \leq size(s_i)$. To see that the relation is strict, consider the literal $b_t \cdot \theta'_i$. The proof is by cases. If $b_t \cdot \theta'_i \in s_i$, then the size of s must be smaller than that of s_i because its counterpart in s ($b_t \cdot \theta$) does not appear in s and the lgg never substitutes a term by one of greater size. If $b_t \cdot \theta'_i \notin s_i$, then either s_i contains an extra literal (thus $size(s) \lneq size(s_i)$), or $[s_x, c_x]$ can not be a counterexample. ■

As a direct consequence, we obtain that whenever a counterexample is appended to the end of the sequence S, it is because there is no other element in S capturing a clause in $U(T)$ that is also captured by $[s_x, c_x]$.

Lemma 4. *Let $[s_1, c_1]$ and $[s_2, c_2]$ be two full multi-clauses. Let $[s, c]$ be any legal pairing between them. If $[s, c]$ captures a clause $ineq(s_t), s_t \rightarrow b_t$, then the following holds:*

1. *Both $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \rightarrow b_t$.*
2. *At least one of $[s_1, c_1]$ or $[s_2, c_2]$ captures $ineq(s_t), s_t \rightarrow b_t$.*

Proof. By assumption, $ineq(s_t), s_t \rightarrow b_t$ is captured by $[s, c]$, i.e., there is a θ such that $s_t \cdot \theta \subseteq s$, $ineq(s_t) \cdot \theta \subseteq ineq(s)$ and $b_t \cdot \theta \in c$. This implies that if t, t' are two distinct terms of s_t, then $t \cdot \theta$ and $t' \cdot \theta$ are distinct terms appearing in s. Let σ be the 1-1 legal matching inducing the pairing. The antecedent s is defined to be $lgg_{|\sigma}(s_1, s_2)$, and therefore there exist substitutions θ_1 and θ_2 such that $s \cdot \theta_1 \subseteq s_1$ and $s \cdot \theta_2 \subseteq s_2$.

Condition 1. We claim that $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \rightarrow b_t$ via $\theta \cdot \theta_1$ and $\theta \cdot \theta_2$, respectively. Notice that $s_t \cdot \theta \subseteq s$, and therefore $s_t \cdot \theta \cdot \theta_1 \subseteq s \cdot \theta_1$. Since $s \cdot \theta_1 \subseteq s_1$, we obtain $s_t \cdot \theta \cdot \theta_1 \subseteq s_1$. The same holds for s_2. It remains to show that $ineq(s_t) \cdot \theta \cdot \theta_1 \subseteq ineq(s_1)$ and $ineq(s_t) \cdot \theta \cdot \theta_2 \subseteq ineq(s_2)$. Observe that all top-level terms appearing in s also appear as one entry of the matching σ, because otherwise they could not have survived. Further, since σ is legal, all subterms of terms of s also appear as an entry in σ. Let t, t' be any distinct terms appearing in s_t. Since $s_t \cdot \theta \subseteq s$ and σ includes all terms appearing in s,

the distinct terms $t \cdot \theta$ and $t' \cdot \theta$ appear as the lgg of distinct entries in σ. These entries have the form $[t \cdot \theta \cdot \theta_1 \texttt{ - } t \cdot \theta \cdot \theta_2 \texttt{ => } t \cdot \theta]$, since $lgg(t \cdot \theta \cdot \theta_1, t \cdot \theta \cdot \theta_2) = t \cdot \theta$. Since σ is 1-1, we know that $t \cdot \theta \cdot \theta_1 \neq t' \cdot \theta \cdot \theta_1$ and $t \cdot \theta \cdot \theta_2 \neq t' \cdot \theta \cdot \theta_2$.

Condition 2. By hypothesis, $b_t \cdot \theta \in c$ and c is defined to be $lgg_{|\sigma}(s_1, c_2) \cup lgg_{|\sigma}(c_1, s_2) \cup lgg_{|\sigma}(c_1, c_2)$. Observe that all these lggs share the same table, so the same pairs of terms will be mapped into the same expressions. Observe also that the substitutions θ_1 and θ_2 are defined according to this table, so that if any literal $l \in lgg_{|\sigma}(c_1, \cdot)$, then $l \cdot \theta_1 \in c_1$. Equivalently, if $l \in lgg_{|\sigma}(\cdot, c_2)$, then $l \cdot \theta_2 \in c_2$. Therefore we get that if $b_t \cdot \theta \in lgg_{|\sigma}(c_1, \cdot)$, then $b_t \cdot \theta \cdot \theta_1 \in c_1$ and if $b_t \cdot \theta \in lgg_{|\sigma}(\cdot, c_2)$, then $b_t \cdot \theta \cdot \theta_2 \in c_2$. Now, observe that in any of the three possibilities for c, one of c_1 or c_2 is included in the $lgg_{|\sigma}$. Thus it is the case that either $b_t \cdot \theta \cdot \theta_1 \in c_1$ or $b_t \cdot \theta \cdot \theta_2 \in c_2$. Since both $[s_1, c_1]$ and $[s_2, c_2]$ cover $ineq(s_t), s_t \rightarrow b_t$, one of $[s_1, c_1]$ or $[s_2, c_2]$ captures $ineq(s_t), s_t \rightarrow b_t$. ■

It is crucial for Lemma 4 that the pairing involved is *legal*. It is indeed possible for a *non-legal* pairing to capture some clause that is not even covered by some of its originating multi-clauses.

Lemma 5. *Every time the algorithm is about to make an equivalence query, it is the case that every multi-clause in S captures at least one of the clauses of $U(T)$ and every clause of $U(T)$ is captured by at most one multi-clause in S.*

Proof (Sketch). All counterexamples included in S are full positive multi-clauses. Therefore, every $[s, c]$ in S captures some clause of $U(T)$. An induction on the number of iterations of the main loop in line 2 of the learning algorithm shows that no two different multi-clauses in S capture the same clause of $U(T)$. ■

Recall that m' stands for the number of clauses in the transformation $U(T)$. Lemma 5 provides us with the bound m' on $|S|$ required to guarantee termination of the algorithm. Counting carefully the number of queries made in every procedure we arrive to our main result.

Theorem 1. *The algorithm exactly identifies range restricted Horn expressions making $O(m'st^a)$ equivalence queries and $O(m's^2t^ae_t^{a+1} + m'^2s^2t^{2a+k})$ membership queries. The running time is polynomial in the number of membership queries.*

5 Fully Inequated Range Restricted Horn Expressions

Clauses of this class can contain a new type of literal, that we call *inequation* or *inequality* and has the form $t \neq t'$, where both t and t' are terms. Inequated clauses may contain any number of inequalities in its antecedent. Let s be any conjunction of atoms and inequations. Then, s^p denotes the conjunction of atoms in s and $s^{\neq}$ the conjunction of inequalities in s. That is $s = s^p \wedge s^{\neq}$. We say s is *completely inequated* if $s^{\neq}$ contains all possible inequations between distinct terms in s^p, i.e., if $s^{\neq} = ineq(s^p)$. A clause $s \rightarrow b$ is completely inequated iff s is. A multi-clause $[s, c]$ is completely inequated iff s is. A *fully inequated range*

restricted Horn expression is a conjunction of fully inequated range restricted clauses.

Looking at the way the transformation $U(T)$ described in Section 2.2 is used in the proof of correctness, the natural question of what happens when the target expression is already fully inequated (and $T = U(T)$) arises. We will see that the algorithm presented in Figure 2 has to be slightly modified in order to achieve learnability of this class.

The first modification is in the minimisation procedure. It can be the case that after generalising or dropping some terms (as it is done in the two stages of the minimisation procedure), the result of the operation is not fully inequated. More precisely, there may be superfluous inequalities that involve terms not appearing in the atoms of the counterexample's antecedent. These should be eliminated from the counterexample, yielding a fully inequated minimised counterexample.

Given a matching σ and two multi-clauses $[s_x, c_x]$ and $[s_i, c_i]$, its pairing $[s, c]$ is computed in the new algorithm as: $s = ineq(lgg_{|\sigma}(s_x^p, s_i^p)) \cup lgg_{|\sigma}(s_x^p, s_i^p)$ and $c = lgg_{|\sigma}(s_x^p, c_i) \cup lgg_{|\sigma}(c_x, s_i^p) \cup lgg_{|\sigma}(c_x, c_i)$. Notice that inequations in the antecedents are ignored. The pairing is computed only for the atomic information, and finally the fully inequated pairing is constructed by adding all the inequations needed. This can be done safely because the algorithm only deals with fully inequated clauses. The proof of correctness is very similar to the one presented here. Complete details and proof can be found in [AK00a].

Theorem 2. *The modified algorithm exactly identifies fully inequated range restricted Horn expressions making $O(mst^a)$ calls to the equivalence oracle and $O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$ to the membership oracle. The running time is polynomial in the number of membership queries.*

6 Conclusions

The paper introduced a new algorithm for learning range restricted Horn expressions ($RRHE$) and established the learnability of fully inequated range restricted Horn expressions ($FIRRHE$). The structure of the algorithm is similar to previous ones, but it uses carefully chosen operations that take advantage of the structure of functional terms in examples. This in turn leads to an improvement of worst case bounds on the number of queries required, which is one of the main contributions of the paper. The following table contains the results obtained in [Kha99b] and in this paper.

	Class	$EntEQ$	$EntMQ$
Result in [Kha99b]	$RRHE$	$O(mst^{t+a})$	$O(ms^2t^{t+a}e_t^{a+1} + m^2s^2t^{3t+2a})$
Our result	$RRHE$	$O(mst^{t+a})$	$O(ms^2t^{t+a}e_t^{a+1} + m^2s^2t^{2t+k+2a})$
Our result	$FIRRHE$	$O(mst^a)$	$O(ms^2t^ae_t^{a+1} + m^2s^2t^{2a+k})$

Note that for $RRHE$ the exponential dependence on the number of terms is reduced from t^{3t} to t^{2t+k}. A more dramatic improvement is achieved for

$FIRRHE$ where the comparable factor is t^k so that the algorithm is polynomial in the number of terms t though still exponential in the number of variables k. This may be significant as in many cases, while inequalities are not explicitly written, the intention is that different terms denote different objects. The reduction in the number of queries goes beyond worst case bounds. The restriction that pairings are both basic *and* agree with the lgg table is quite strong and reduces the number of pairings and hence queries. This is not reflected in our analysis but we believe it will make a difference in practice. Similarly, the bound $m' \leq mt^t$ on $|U(T)|$ is quite loose, as a large proportion of partitions will be discarded if T includes functional structure.

Finally, the use of lgg results in a more direct and natural algorithm. Moreover, it may help understand the relations between previous algorithms based on lgg [Ari97, RT98, RS98] and algorithms based on direct products [Kha99a]. We hope that this can lead to further understanding and better algorithms for the problem.

References

[AK00a] M. Arias and R. Khardon. Learning Inequated Range Restricted Horn Expressions. Technical Report EDI-INF-RR-0011, Division of Informatics, University of Edinburgh, March 2000.

[AK00b] M. Arias and R. Khardon. A New Algorithm for Learning Range Restricted Horn Expressions. Technical Report EDI-INF-RR-0010, Division of Informatics, University of Edinburgh, March 2000.

[Ari97] Hiroki Arimura. Learning acyclic first-order Horn sentences from entailment. In *Proceedings of the International Conference on ALT*, Sendai, Japan, 1997. Springer-Verlag. LNAI 1316.

[Coh95a] W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.

[Coh95b] W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.

[FP93] M. Frazier and L. Pitt. Learning from entailment: An application to propositional Horn sentences. In *Proceedings of the International Conference on Machine Learning*, pages 120–127, Amherst, MA, 1993. Morgan Kaufmann.

[Kha99a] R. Khardon. Learning function free Horn expressions. *Machine Learning*, 37:241–275, 1999.

[Kha99b] R. Khardon. Learning range restricted Horn expressions. In *Proceedings of the Fourth European Conference on Computational Learning Theory*, pages 111–125, Nordkirchen, Germany, 1999. Springer-verlag. LNAI 1572.

[Kha00] Roni Khardon. Learning horn expressions with LOGAN-H. To appear in ICML, 2000.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987.

[MF92] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.

[MR94] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19 & 20:629–680, May 1994.

[Plo70] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[RB92] L. De Raedt and M. Bruynooghe. An overview of the interactive concept-learner and theory revisor CLINT. In S. Muggleton, editor, *Inductive Logic Programming*, pages 163–192. Academic Press, 1992.

[RS98] K. Rao and A. Sattar. Learning from entailment of logic programs with local variables. In *Proceedings of the International Conference on Algorithmic Learning Theory*, Otzenhausen, Germany, 1998. Springer-verlag. LNAI 1501.

[RT98] C. Reddy and P. Tadepalli. Learning first order acyclic Horn programs from entailment. In *International Conference on Inductive Logic Programming*, pages 23–37, Madison, WI, 1998. Springer. LNAI 1446.

[SEMF98] G. Semeraro, F. Esposito, D. Malerba, and N. Fanizzi. A logic framework for the incremental inductive synthesis of datalog theories. In *Proceedings of the International Conference on Logic Program Synthesis and Transformation (LOPSTR'97)*. Springer-Verlag, 1998. LNAI 1463.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.

A Refinement Operator for Description Logics

Liviu Badea[1] and Shan-Hwei Nienhuys-Cheng[2]

[1] AI Lab, National Institute for Research and Development in Informatics
8-10 Averescu Blvd., Bucharest, Romania.
badea@ici.ro
[2] Erasmus University Rotterdam, H9-14, Post Box 1738
3000 DR, Rotterdam, The Netherlands.
cheng@few.eur.nl

Abstract. While the problem of learning logic programs has been extensively studied in ILP, the problem of learning in description logics (DLs) has been tackled mostly by empirical means. Learning in DLs is however worthwhile, since both Horn logic and description logics are widely used knowledge representation formalisms, their expressive powers being incomparable (neither includes the other as a fragment). Unlike most approaches to learning in description logics, which provide bottom-up (and typically overly specific) *least* generalizations of the examples, this paper addresses learning in DLs using downward (and upward) refinement operators. Technically, we construct a *complete* and *proper* refinement operator for the $\mathcal{ALER}$ description logic (to avoid overfitting, we disallow disjunctions from the target DL). Although no *minimal* refinement operators exist for $\mathcal{ALER}$, we show that we can achieve minimality of all refinement steps, except the ones that introduce the $\bot$ concept. We additionally prove that complete refinement operators for $\mathcal{ALER}$ cannot be *locally finite* and suggest how this problem can be overcome by an MDL search heuristic. We also discuss the influence of the Open World Assumption (typically made in DLs) on example coverage.

1 Introduction

The field of machine learning has witnessed an evolution from ad-hoc specialized systems to increasingly more general algorithms and languages. This is not surprising since a learning algorithm aims at improving the behaviour of an *existing* system. And since early systems were quite diverse, the early learning systems were ad-hoc and thus hard to capture in a unified framework. Nevertheless, important progresses were made in the last decade towards learning in very general settings, such as first order logic. Inductive Logic Programming (ILP) deals with learning first order logic programs. Very recently the expressiveness of the target language was extended to prenex conjunctive normal forms [20] by allowing existential quantifiers in the language.

Description Logics (DL), on the other hand, are a different kind of knowledge representation language used for representing structural knowledge and

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 40–59, 2000.

concept hierarchies. They represent a function-free first order fragment allowing a variable-free syntax, which is considered to be important for reasons of readability (the readability of the specifications is an important issue in knowledge representation, where the expressiveness and tractability of the language are only considered together with the simplicity and understandability of the representations).

Description logics are subsets of first order logic containing only unary predicates (called *concepts* and representing sets of objects in the domain) and binary predicates (referred to as *roles* and representing binary relations between domain objects). A number of concept and role *constructors* can be used to express compound concept terms, which are variable-free representations of first-order descriptions.

The research in description logics has concentrated on (practical) algorithms and precise complexity results [9] for testing concept subsumption (inclusion) and consistency, as well as checking the membership of objects (*individuals*) in concepts for various combinations of concept and role constructors. These inference services are particularly useful for managing hierarchical knowledge (description logics usually provide automatic classification of concepts in a concept hierarchy).

Since the expressivities of Horn logic and description logics are incomparable [5], and since one of the main limitations of Horn rules is their inability to model and reason about value restrictions in domain with a rich hierarchical structure, there have been several attempts at combining description logics and (function-free) Horn rules (for example AL-log [10], CARIN [14,15]). Reasoning in such combined languages is in general harder than in the separate languages [15].

While deduction in description logics has been thoroughly investigated and also while learning Horn rules has already reached a mature state (in the field of Inductive Logic Programming), learning DL descriptions from examples has been approached mostly by heuristic means [7,8,12]. For example, LCSLEARN [8] is a bottom-up learning algorithm using least common subsumers (LCS) as generalizations of concepts. It's disjunctive version, LCSLEARNDISJ, is similar to the ILP system GOLEM since LCSs play the role of least general generalizers (LGGs). Bottom-up ILP systems like GOLEM proved quite successful in certain applications (such as determining protein secondary structure), but they usually produce overly specific hypotheses. In Inductive Logic Programming, this drawback was eliminated by top-down systems like FOIL and Progol [16], which use *downward* refinement operators for exploring the space of hypotheses.

This paper aims at going beyond simple LCS-based learning in DLs by constructing complete upward and downward refinement operators for the description logic $\mathcal{ALER}$. This complete refinement operator is used by a more sophisticated learning algorithm to induce DL descriptions from examples.

Since description logics are a fragment of first-order logic, we could in principle translate DL expressions into prenex conjunctive form (PCNF) and use the

PCNF-refinement operator defined by [20] to refine the PCNF encoding of DL expressions.[1]

Example 1. We could obtain the DL definition $Influencial \leftarrow \forall Friend.Influencial$ by the following squence of PCNF refinement steps:
Adding literals:

$\square \rightarrow \forall x I(x)$
$\rightarrow \forall x \forall y \forall z I(x) \vee F(y,z)$
$\rightarrow \forall x \forall y \forall z \forall u (I(x) \vee F(y,z)) \wedge I(u)$
$\rightarrow \forall x \forall y \forall z \forall u \forall v (I(x) \vee F(y,z)) \wedge (I(u) \vee \neg I(v))$

Unifications:

$\rightarrow \forall x \forall z \forall u \forall v (I(x) \vee F(x,z)) \wedge (I(u) \vee \neg I(v))$
$\rightarrow \forall x \forall z \forall v (I(x) \vee F(x,z)) \wedge (I(x) \vee \neg I(v))$
$\rightarrow \forall x \forall z (I(x) \vee F(x,z)) \wedge (I(x) \vee \neg I(z))$

Substitutions:

$\rightarrow \forall x (I(x) \vee F(x,c)) \wedge (I(x) \vee \neg I(c))$, where c is a new constant

e-substitution:

$\rightarrow \exists y \forall x (I(x) \vee F(x,y)) \wedge (I(x) \vee \neg I(y))$

eu-substitution:

$\rightarrow \forall x \exists y (I(x) \vee F(x,y)) \wedge (I(x) \vee \neg I(y))$

However, this straightforward approach has two disadvantages:

- The conversion from DL to PCNF can lead to an (exponential) blow-up of the formulae. Converting everything to clausal form may spoil the structure of the initial DL description and the conversion of the refinement back to DL may affect the readability of the result.
- In fact, since DL descriptions are coarser grained than FOL formulae, some PCNF refinements may not even have a DL counterpart. More precisely, if ρ_{DL} is a DL-refinement operator and ρ a PCNF-refinement operator, then for every $D \in \rho_{DL}(C)$ we have $PCNF(D) \in \rho^*(PCNF(C))$, where $PCNF(C)$ is the PCNF encoding of the DL expression C. However, not every $D' \in \rho(PCNF(C))$ is the PCNF counterpart of a DL formula D: $D' = PCNF(D)$. Worse still, there can be arbitrarily long PCNF refinement chains (of some DL concept) that have no DL correspondence at all. Apparently, the PCNF refinement steps are too fine grained.

To circumvent these problems, we need to develop a refinement operator working directly on DL formulae. But directly refining DL formulae has an even deeper justification than just convenience. In fact, the hypotheses language (in our case a DL) plays the role of *learning bias*, which determines the granularity of the generalization steps. A very fine grained language (like FOL or PCNF) may be unsuitable for generalizing coarser grained DL descriptions.

[1] We need a refinement operator for PCNFs rather than universally quantified clauses because description logic expressions containing existential restrictions introduce existential quantifiers into their first-order encodings.

2 The Learning Problem in Description Logics

In Description Logics, concepts represent classes of objects in the domain of interest, while roles represent binary relations in the same domain.

Complex concepts ($C, D, \ldots$) and roles ($R, Q, S, \ldots$) in the $\mathcal{ALER}$ description logic can be built from atomic concepts (A) and primitive roles (P) using the following concept and the role constructors:

Concept constructor	Syntax	Interpretation
top concept	$\top$	Δ
bottom concept	$\bot$	$\emptyset$
negation of atoms	$\neg A$	$\Delta \setminus A^{\mathcal{I}}$
concept conjunction	$C_1 \sqcap C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
value restriction	$\forall R.C$	$\{x \in \Delta \mid \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
existential restriction	$\exists R.C$	$\{x \in \Delta \mid \exists y.(x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$

Role constructor	Syntax	Interpretation
role conjunction	$R_1 \sqcap R_2$	$R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$

An interpretation $\mathcal{I} = (\Delta, \cdot^{\mathcal{I}})$ consists of a non-empty set Δ (the interpretation domain) and an interpretation function $\cdot^{\mathcal{I}}$ which maps concepts to subsets of Δ and roles to subsets of $\Delta \times \Delta$ according to the relationships in the table above.

A knowledge base $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ has two components: a Tbox (terminology) $\mathcal{T}$ and Abox (assertional component) $\mathcal{A}$. A terminology $\mathcal{T}$ contains a set of definitions (terminological axioms) as below, representing intensional knowledge about classes of objects:

Definition of A	Syntax	Interpretation
sufficient	$A \leftarrow C$	$A^{\mathcal{I}} \supseteq C^{\mathcal{I}}$
necessary	$A \rightarrow C$	$A^{\mathcal{I}} \subseteq C^{\mathcal{I}}$
necessary and sufficient	$A = C$	$A^{\mathcal{I}} = C^{\mathcal{I}}$

The Abox $\mathcal{A}$ contains extensional information in the form of membership assertions stating that specific individuals are instances of certain concepts or are involved in relations with other individuals as tuples of certain roles.

Assertion	Syntax	Interpretation
concept instance	$C(a)$	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
role tuple	$R(a,b)$	$(a,b)^{\mathcal{I}} \in R^{\mathcal{I}}$

An interpretation satisfies a (terminological or assertional) axiom iff the conditions in the tables above hold. (The interpretation function maps individuals, such as a and b, to domain elements such that the *unique names assumption* is satisfied: $a^{\mathcal{I}} \neq b^{\mathcal{I}}$ if $a \neq b$.)

An interpretation that satisfies all the axioms of the knowledge base $\mathcal{K}$ is a *model* of $\mathcal{K}$.

Description logics represent a first-order logic fragment written in a variable-free syntax. The first-order encoding of $\mathcal{ALER}$ descriptions is given below.

C	$C(x)$
$\top$	$true$
$\bot$	$false$
$\neg A$	$\neg A(x)$
$C_1 \sqcap C_2$	$C_1(x) \wedge C_2(x)$
$\forall R.C$	$\forall y.R(x,y) \rightarrow C(y)$
$\exists R.C$	$\exists y.R(x,y) \wedge C(y)$

R	$R(x,y)$
$R_1 \sqcap R_2$	$R_1(x,y) \wedge R_2(x,y)$

Definition	FOL encoding
$A \leftarrow C$	$\forall x.\ A(x) \leftarrow C(x)$
$A \rightarrow C$	$\forall x.\ A(x) \rightarrow C(x)$
$A = C$	$\forall x.\ A(x) \leftrightarrow C(x)$

Assertion	FOL encoding
$C(a)$	$C(a)$
$R(a,b)$	$R(a,b)$

Description logics trade expressive power for more efficient (dedicated) inference services, as well as for an increased readability. Unlike FOL reasoners, DLs provide *complete* decision algorithms (which are guaranteed to terminate and whose complexity is tightly controlled) for the following (deductive) reasoning services:

- *KB-satisfiability*: $\mathcal{K}$ is satisfiable iff it admits a model.
- *concept satisfiability*: C is satisfiable w.r.t. $\mathcal{K}$ iff $\mathcal{K}$ admits a model $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$.
- *concept equivalence*: C and D are equivalent w.r.t. $\mathcal{K}$, $\mathcal{K} \models C \equiv D$, iff $C^{\mathcal{I}} = D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$.
- *concept subsumption*: C is subsumed[2] by D w.r.t. $\mathcal{K}$, $\mathcal{K} \models C \rightarrow D$, iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ for every model $\mathcal{I}$ of $\mathcal{K}$. (We sometimes also write $C \sqsubseteq D$ instead of $C \rightarrow D$ to emphasize that subsumption is a generality order.)
  C is *strictly subsumed* by D, $C \sqsubset D$ iff $C \sqsubseteq D$ and $C \not\equiv D$.
- *instance checking*: a is an instance of C, $\mathcal{K} \models C(a)$, iff the assertion $C(a)$ is satisfied in every model of $\mathcal{K}$.

All the above *deductive* inference services can be reduced to $\mathcal{ALCR}$ KB-satisfiability [6].

Typically, terminological axioms have been used in DLs mainly for stating constraints (in the form of necessary definitions) on concepts. In our learning framework however, we also need sufficient definitions for classifying individuals as being instances of a given concept. Although theoretical work has addressed the issue of very general terminological axioms (such as general inclusions [6]), most existing DL implementations do not allow sufficient definitions in the TBox. However, we can "simulate" a set of sufficient definitions $\{A \leftarrow C_1, \ldots, A \leftarrow C_n\}$ in a DL with disjunctions by using the necessary and sufficient definition $A = C_1 \sqcup \ldots \sqcup C_n \sqcup A'$, where A' is a new concept name.

In this paper we aim at endowing DLs with *inductive* inference services as well. The *learning problem* in DLs can be stated as follows.

[2] Concept subsumption in description logics should not be confused with clause subsumption in ILP and Automated Reasoning.

Definition 1. *Let $\mathcal{K} = \langle \mathcal{T}, \mathcal{A} \rangle$ be a DL knowledge base. A subset $\mathcal{A}' \subseteq \mathcal{A}$ of the assertions $\mathcal{A}$ can be viewed as (ground)* examples *of the target concept A:*

$$\mathcal{A}' = \{A(a_1), A(a_2), \ldots, \neg A(b_1), \neg A(b_2), \ldots\}.$$

The DL learning problem consists in inducing a set of concept definitions for A: $\mathcal{T}'' = \{A \leftarrow C_1, A \leftarrow C_2, \ldots\}$ that covers the examples $\mathcal{A}'$, i.e.

$$\langle \mathcal{T} \cup \mathcal{T}'', \mathcal{A} \setminus \mathcal{A}' \rangle \models \mathcal{A}'.$$

In other words, we replace the specific examples $\mathcal{A}'$ from the Abox with more general Tbox definitions $\mathcal{T}''$. Note that, after learning, the knowledge base will be $\langle \mathcal{T} \cup \mathcal{T}'', \mathcal{A} \rangle$, which is equivalent with $\langle \mathcal{T} \cup \mathcal{T}'', \mathcal{A} \setminus \mathcal{A}' \rangle$, since the latter is supposed to cover the examples $\mathcal{A}'$.

Alternatively, we could consider a more general setting in which Tbox definitions of A can also serve as examples. In this case, the examples are a knowledge base $\langle \mathcal{T}', \mathcal{A}' \rangle$ such that

$\mathcal{T}' = \{A \leftarrow D_1, A \leftarrow D_2, \ldots\} \subseteq \mathcal{T}$

$\mathcal{A}' = \{A(a_1), A(a_2), \ldots, \neg A(b_1), \neg A(b_2), \ldots\} \subseteq \mathcal{A}$

and the learning problem consists in inducing a set of concept definitions for A $\mathcal{T}'' = \{A \leftarrow C_1, A \leftarrow C_2, \ldots\}$ that cover the examples $\langle \mathcal{T}', \mathcal{A}' \rangle$, i.e.

$$\langle (\mathcal{T} \setminus \mathcal{T}') \cup \mathcal{T}'', \mathcal{A} \setminus \mathcal{A}' \rangle \models \langle \mathcal{T}', \mathcal{A}' \rangle.$$

(The knowledge base after learning $\langle \mathcal{T} \cup \mathcal{T}'', \mathcal{A} \rangle$ is equivalent with $\langle (\mathcal{T} \setminus \mathcal{T}') \cup \mathcal{T}'', \mathcal{A} \setminus \mathcal{A}' \rangle$.) Note that, in this setting, $\langle \mathcal{T} \setminus \mathcal{T}', \mathcal{A} \setminus \mathcal{A}' \rangle$ plays the role of background knowledge.

The learning problem thus formulated is very similar to the standard setting of Inductive Logic Programming. The main differences consist in the different expressivities of the target hypotheses spaces and the *Open World Assumption* (OWA) adopted by DLs (as opposed to the Closed World Assumption usually made in (Inductive) Logic Programming).

3 $\mathcal{ALER}$ Refinement Operators

While Inductive Logic Programming (ILP) systems learn logic programs from examples, a DL-learning system should learn DL descriptions from Abox instances. Both types of learning systems traverse large spaces of hypotheses in an attempt to come up with an optimal consistent hypothesis as fast as possible. Various heuristics can be used to guide this search, for instance based on information gain, MDL [16], etc. A simple search algorithm (even a complete and non-redundant one) would not do, unless it allows for a flexible traversal of the search space, based on an external heuristic. Refinement operators allow us to decouple the heuristic from the search algorithm. Downward (upward) refinement operators construct specializations (generalizations) of hypotheses and are usable in a top-down (respectively bottom-up) search of the space of hypotheses.

Definition 2. *A* downward (upward) refinement operator *is a mapping* ρ *from hypotheses to sets of hypotheses (called refinements) which produces only specializations (generalizations), i.e.* $H' \in \rho(H)$ *implies* $H \models H'$ *(respectively* $H' \models H$*). (We shall sometimes also write* $H \rightsquigarrow H'$ *instead of* $H' \in \rho(H)$*.)*

If the hypotheses are *sufficient definitions*, then $A \leftarrow C_1$ is more general than (entails) $A \leftarrow C_2$ iff C_1 subsumes (is more general than) C_2 ($C_1 \sqsupseteq C_2$).

In ILP the *least general generalization (lgg)* of two clauses is the least general *clause* that subsumes both. Note that although such an *lgg* is more general than conjunction, taking the *lgg* is justified by the fact that the conjunction of clauses is not a clause and would only overfit the input clauses (i.e. wouldn't perform any generalization leap).

Similarly with ILP, we define the *lgg* of two sufficient definitions as

$$lgg(A \leftarrow C_1, A \leftarrow C_2) = A \leftarrow lcs(C_1, C_2),$$

where $lcs(C_1, C_2)$ is the *least common subsumer* [7] of the DL concepts C_1 and C_2. In order to avoid overfitting and allow generalization leaps, the DL used as a target language should not provide concept disjunctions[3], since the *lcs* would otherwise reduce to concept disjunction.

For hypotheses representing *necessary definitions*, $A \rightarrow C_1$ is more general than (entails) $A \rightarrow C_2$ iff C_1 is subsumed by (is more specific than) C_2 ($C_1 \sqsubseteq C_2$). The *lgg* of two necessary definitions reduces to concept conjunction

$$lgg(A \rightarrow C_1, A \rightarrow C_2) = A \rightarrow C_1 \sqcap C_2,$$

while their *most general specialization (mgs)* is

$$mgs(A \rightarrow C_1, A \rightarrow C_2) = A \rightarrow lcs(C_1, C_2).$$

While sufficient definitions ($A \leftarrow C_{suff}$) represent lower bounds on the target concept A, necessary definitions ($A \rightarrow C_{nec}$) represent upper bounds on A, i.e. $C_{suff} \sqsubseteq A \sqsubseteq C_{nec}$.

Note that a downward refinement operator on concept descriptions C (going from $\top$ to $\bot$) induces a downward refinement operator on sufficient definitions $A \leftarrow C$ and an upward refinement operator on necessary definitions $A \rightarrow C$.

But unlike necessary definitions $A \rightarrow C_1, \ldots, A \rightarrow C_n$, whose conjunction can be expressed as a single necessary definition $A \rightarrow C_1 \sqcap \ldots \sqcap C_n$, sufficient definitions like $A \leftarrow C_1, \ldots, A \leftarrow C_n$ cannot be expressed as a single sufficient definition unless the language allows concept disjunction: $A \leftarrow C_1 \sqcup \ldots \sqcup C_n$.[4]

In the following, we construct a *complete* downward refinement operator for $\mathcal{ALER}$ concepts.

[3] this justifies our choice of the target language $\mathcal{ALER}$ instead of the more expressive $\mathcal{ALCR}$, which allows concept disjunctions.

[4] Of course, such necessary definitions could be approximated by $A \leftarrow lcs(C_1, \ldots, C_n)$, but this is more general than the conjunction of the original definitions.

Definition 3. *A downward refinement operator ρ on a set of concepts ordered by the subsumption relationship $\sqsupseteq$ is called*

- (locally) finite *iff $\rho(C)$ is finite for every hypothesis C.*
- complete *iff for all C and D, if C is strictly more general than D ($C \sqsupset D$), then $\exists E \in \rho^*(C)$ such that $E \equiv D$.*
- weakly complete *iff $\rho^*(\top)$ = the entire set of hypotheses.*
- redundant *iff there exists a refinement chain[5] from C_1 to D not going through C_2 and a refinement chain from C_2 to D not going through C_1.*
- minimal *iff for all C, $\rho(C)$ contains only downward covers[6] and all its elements are incomparable.*
- proper *iff for all C and D, $D \in \rho(C)$ entails $D \sqsubset C$ (or, equivalently, $D \not\equiv C$).*

(For defining the corresponding properties of upward *refinement operators, we simply need to replace the generality order $\sqsupseteq$ by its dual $\sqsubseteq$.)*

We first construct a complete but *non-minimal* (and thus highly redundant[7]) refinement operator for which the completeness proof is rather simple. Subsequently, we will modify this operator (while preserving its completeness) to reduce its non-minimality.

The refinement operator ρ_0 is given by the following refinement rules (recall that $C \rightsquigarrow D$ means $D \in \rho_0(C)$, which entails the fact that C subsumes D, $C \sqsupseteq D$):

Refinement rules of ρ_0

$[Lit]$ $C \rightsquigarrow C \sqcap L$ with L a *DL-literal* (to be defined below)
$[\exists C]$ $C \sqcap \exists R.C_1 \rightsquigarrow C \sqcap \exists R.C_2$ if $C_1 \rightsquigarrow C_2$
$[\exists R]$ $C \sqcap \exists R_1.D \rightsquigarrow C \sqcap \exists R_2.D$ if $R_1 \rightsquigarrow R_2$
$[\exists\exists]$ $C \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2 \rightsquigarrow C \sqcap \exists(R_1 \sqcap R_2).(C_1 \sqcap C_2)$
$[\forall C]$ $C \sqcap \forall R.C_1 \rightsquigarrow C \sqcap \forall R.C_2$ if $C_1 \rightsquigarrow C_2$
$[\forall R]$ $C \sqcap \forall R_1.D \rightsquigarrow C \sqcap \forall R_2.D$ if $R_2 \rightsquigarrow R_1$

$[PR]$ $R \rightsquigarrow R \sqcap P$ with P a primitive role.

[5] A *refinement chain* from C to D is a sequence $C_0, C_1, \ldots, C_n$ of hypotheses such that $C = C_0, C_1 \in \rho(C_0), C_2 \in \rho(C_1), \ldots, C_n \in \rho(C_{n-1}), D = C_n$. Such a refinement chain does not 'go through' E iff $E \neq C_i$ for $i = 0, \ldots, n$.

[6] D is a *downward cover* of C iff C is more general than D ($C \sqsupset D$) and no E satisfies $C \sqsupset E \sqsupset D$.

[7] There are two possible sources of *redundancy* in a refinement operator:
– non-minimal refinement steps, and
– the possibility of reaching a hypothesis from two different incomparable hypotheses.
Here we refer to the first type of redundancy, which can be eliminated by disallowing non-minimal steps. (The second type of redundancy can be eliminated using the methods from [3,4].)

The refinement rules above apply to concepts in *$\mathcal{ALER}$-normal form*, which can be obtained for a concept by applying the following identities as rewrite rules[8] left-to-right until they are no longer applicable:

$$
\begin{array}{ll}
[\forall\forall] & \forall R.C \sqcap \forall R.D = \forall R.(C \sqcap D) \\
[\forall\forall R] & \forall R.C \sqcap \forall Q.D = \forall R.(C \sqcap D) \sqcap \forall Q.D \quad \text{if } R \sqsubset Q \\
[\exists\forall] & \exists R.C \sqcap \forall Q.D = \exists R.(C \sqcap D) \sqcap \forall Q.D \quad \text{if } R \sqsubseteq Q \\
\forall R.\top = \top & \exists R.\bot = \bot \\
C \sqcap \neg C = \bot & C \sqcap \top = C \qquad C \sqcap \bot = \bot
\end{array}
$$

For example, the normal form of $\forall(P_1 \sqcap P_2).A_1 \sqcap \forall P_1.\neg A_1 \sqcap \forall P_1.A_2 \sqcap \exists P_1.A_3$ is $\forall(P_1 \sqcap P_2).\bot \sqcap \forall P_1.(\neg A_1 \sqcap A_2) \sqcap \exists P_1.(\neg A_1 \sqcap A_2 \sqcap A_3)$.

Definition 4. *In the refinement rule* $[Lit]$*, a* DL-literal *is either*

- *an atom* (A)*, the negation of an atom* $(\neg A)$*,*
- *an existential restriction* $\exists P.\top$ *for a primitive role* P*, or*
- *a value restriction* $\forall \prod_{i=1}^{n} P_i.L'$ *with* L' *a* DL-literal*, where* $\{P_1, .., P_n\}$ *is the set of all primitive roles occurring in the knowledge base.*

An example of a ρ_0-chain: $\top \overset{[Lit]}{\rightsquigarrow} A_1 \overset{[Lit]}{\rightsquigarrow} A_1 \sqcap \forall(P_1 \sqcap P_2).A_2 \overset{[Lit]}{\rightsquigarrow} A_1 \sqcap \forall(P_1 \sqcap P_2).A_2 \sqcap \exists P_2.\top \overset{[\exists R]}{\rightsquigarrow} A_1 \sqcap \forall(P_1 \sqcap P_2).A_2 \sqcap \exists P_2.A_3 \overset{[\forall R]}{\rightsquigarrow} A_1 \sqcap \forall P_1.A_2 \sqcap \exists P_2.A_3 \overset{[Lit]}{\rightsquigarrow} A_1 \sqcap \forall P_1.A_2 \sqcap \exists P_2.A_3 \sqcap \exists P_2.\top \overset{[\exists R]}{\rightsquigarrow} A_1 \sqcap \forall P_1.A_2 \sqcap \exists P_2.A_3 \sqcap \exists P_2.A_4 \overset{[\exists\exists]}{\rightsquigarrow} A_1 \sqcap \forall P_1.A_2 \sqcap \exists P_2.(A_3 \sqcap A_4)$.

The above definition of DL-literal can be explained as follows. The addition of new *"DL-literals"* in the $[Lit]$ rule should involve not just atoms and negations of atoms (i.e. ordinary literals), but also existential and value restrictions. For minimality, these have to be *most general* w.r.t. the concept to be refined C (as well as *non-redundant*, if possible).

The *most general existential restrictions* take the form $\exists P.\top$ for a primitive role P. But if P already occurs in some other existential restriction on the "top level" of C (i.e. $C = C' \sqcap \exists R_1.C_1$ such that $P \sqsupseteq R_1$, or, in other words, $R_1 = R_1' \sqcap P$), then $\exists P.\top$ is redundant w.r.t. C. This is due to the following result.

Proposition 1. $\exists R_1.C_1 \sqsubseteq \exists R_2.C_2$ *if* $R_1 \sqsubseteq R_2$ *and* $C_1 \sqsubseteq C_2$.

Consequently, the restriction to be added $\exists R_2.C_2$ is redundant w.r.t. some other existential restriction $\exists R_1.C_1$ (i.e. $\exists R_1.C_1 \sqcap \exists R_2.C_2 \equiv \exists R_1.C_1$) if $C_1 \sqsubseteq C_2$ and $R_1 \sqsubseteq R_2$.

But disallowing the addition of $\exists P.\top$ to C in cases in which some $\exists R_1.C_1$ with $R_1 \sqsubseteq P$ already occurs in C (which would ensure the *properness* of ρ_0) would unfortunately also lead to the *incompleteness* of ρ_0. For example, it would be

[8] Under associativity, commutativity and idempotence of $\sqcap$.

impossible to reach $\exists P.A_1 \sqcap \exists P.A_2$ as a refinement of $\exists P.A_1$, because the first step in the following chain of refinements[9]:

$$\exists P.A_1 \overset{[Lit]}{\rightsquigarrow} \exists P.A_1 \sqcap \exists P.\top \overset{[\exists C]}{\rightsquigarrow} \exists P.A_1 \sqcap \exists P.A_2$$

would fail, due to the redundancy of $\exists P.\top$.

$\exists P.\top$ is redundant because it is too general. Maybe we could try to directly add something more specific, but non-redundant (like $\exists P.A_2$ in the example above). However, determining the most general *non-redundant* existential restrictions is complicated. We will therefore allow the refinement operator ρ to be improper (i.e. produce refinements $D \in \rho(C)$ that are equivalent to C[10]), but – in order to obtain proper refinements – we will successively apply ρ until a strict refinement (i.e. some $D \sqsubset C$) is produced. The resulting refinement operator ρ^{cl} (the "closure" of ρ) will be proper, by construction. More precisely:

Definition 5. *$D \in \rho^{cl}(C)$ iff there exists a refinement chain of ρ:*

$$\boxed{C} \overset{\rho}{\rightsquigarrow} C_1 \overset{\rho}{\rightsquigarrow} C_2 \overset{\rho}{\rightsquigarrow} \ldots \overset{\rho}{\rightsquigarrow} \boxed{C_n = D},$$

such that $C_i \equiv C$ for $i = 1, \ldots, n-1$ and $C_n \sqsubset C$.

In the above-mentioned example, we have the following refinement chain of ρ_0:

$$C = \exists P.A_1 \overset{[Lit]}{\rightsquigarrow} C_1 = \exists P.A_1 \sqcap \exists P.\top \overset{[\exists C]}{\rightsquigarrow} C_2 = \exists P.A_1 \sqcap \exists P.A_2$$

for which $C \equiv C_1$ and $C \sqsupset C_2$. Therefore, $C_2 \in \rho_0^{cl}(C)$ is a one-step refinement of the "closure" ρ_0^{cl}.

For determining the *most general value restrictions*, we consider the dual of Proposition 1:

Proposition 2. *$\forall R_1.C_1 \sqsubseteq \forall R_2.C_2$ if $R_1 \sqsupseteq R_2$* and *$C_1 \sqsubseteq C_2$.*

(In fact, we can prove an even stronger result: W.r.t. an $\mathcal{ALER}$ knowledge base $\mathcal{K}$ with no necessary definitions, $\mathcal{K} \models \forall R_1.C_1 \sqsubseteq \forall R_2.C_2$ iff $\mathcal{K} \models R_1 \sqsupseteq R_2$ *and*

[9] which is the only chain that can lead from $\exists P.A_1$ to $\exists P.A_1 \sqcap \exists P.A_2$

[10] The specific syntactic form of D is important in this case. In order to preserve completeness, we *disallow* the use of the following *redundancy elimination rules* (which will be used only for simplifying the result returned to the user):

$[\forall red]$ $\forall R_1.C_1 \sqcap \forall R_2.C_2 = \forall R_1.C_1$ if $R_1 \sqsupseteq R_2$ *and* $C_1 \sqsubseteq C_2$
$[\exists red]$ $\exists R_1.C_1 \sqcap \exists R_2.C_2 = \exists R_1.C_1$ if $R_1 \sqsubseteq R_2$ and $C_1 \sqsubseteq C_2$

For example, their use would disallow obtaining $\exists P.A_1 \sqcap \exists P.A_2$ from $\exists P.A_1$:

$$\exists P.A_1 \overset{[Lit]}{\rightsquigarrow} \exists P.A_1 \sqcap \exists P.\top \overset{[\exists C]}{\rightsquigarrow} \exists P.A_1 \sqcap \exists P.A_2$$

because $\exists P.A_1 \sqcap \exists P.\top$ would be simplified by the $[\exists red]$ redundancy elimination rule to $\exists P.A_1$, thereby making the second step ($[\exists C]$) inapplicable.

$\mathcal{K} \models C_1 \sqsubseteq C_2$. Thus, the restriction to be added $\forall R_2.C_2$ is non-redundant w.r.t. $\forall R_1.C_1$ (i.e. $\forall R_1.C_1 \sqcap \forall R_2.C_2 \not\equiv \forall R_1.C_1$) iff $R_1 \not\sqsupseteq R_2$ or $C_1 \not\sqsubseteq C_2$.)

Formally, the most general value restrictions take the form $\forall R.\top$. But unfortunately, such value restrictions are redundant due to the identity $\forall R.\top = \top$. Less redundant value restrictions are $\forall R.L'$, where L' is a DL-literal. Note that R in $\forall R.L'$ cannot be just a primitive role P, since for example $\forall(P \sqcap R').L'$ is more general than (subsumes) $\forall P.L'$. With respect to R, the most general value restriction thus involves a conjunction of all primitive roles in the knowledge base $\forall \prod_{i=1}^{n} P_i.L'$, but unfortunately this is, in general, also redundant. As shown above, redundancy can be eliminated by considering the "closure" ρ_0^{cl} of ρ_0.

3.1 Reducing Non-minimality

It is relatively easy to show that the refinement operator ρ_0 (as well as its closure ρ_0^{cl}) is *complete*. However, it is *non-minimal* (and thereby highly redundant), due to the following $\mathcal{ALER}$ relationships:

$$\exists R.C_1 \sqcap \exists R.C_2 \sqsupseteq \exists R.(C_1 \sqcap C_2), \tag{1}$$

$$\forall R.C_1 \sqcap \forall R.C_2 = \forall R.(C_1 \sqcap C_2). \tag{2}$$

- (1) suggests that, for reasons of minimality, we should not allow in $[\exists C]$ C_1 inside $\exists R.C_1$ to be refined by literal additions ($[Lit]$) to $C_1 \sqcap L$:

  $$C \sqcap \exists R.C_1 \overset{[\exists C]}{\rightsquigarrow} C \sqcap \exists R.(C_1 \sqcap L),$$

  because this single-step refinement could also be obtained with the following sequence of smaller steps (the $[\exists\top]$ step is defined below):

  $$C \sqcap \exists R.C_1 \overset{[Lit]}{\rightsquigarrow} C \sqcap \exists R.C_1 \sqcap \exists R.\top \overset{[\exists\top]}{\rightsquigarrow} C \sqcap \exists R.C_1 \sqcap \exists R.L \overset{[\exists\exists]}{\rightsquigarrow} C \sqcap \exists R.(C_1 \sqcap L).$$

  In other words, instead of directly refining $\exists R.C_1$ to $\exists R.(C_1 \sqcap L)$, we first add $\exists R.L$, and then merge $\exists R.C_1$ and $\exists R.L$ to $\exists R.(C_1 \sqcap L)$ using $[\exists\exists]$ (the refinement step being justified by (1)).

- Similarly, (2) suggests that, for reasons of minimality, we should disallow in $[\forall C]$ C_1 inside $\forall R.C_1$ to be refined by literal additions ($[Lit]$) to $C_1 \sqcap L$:

  $$C \sqcap \forall R.C_1 \overset{[\forall C]}{\rightsquigarrow} C \sqcap \forall R.(C_1 \sqcap L),$$

  because this single-step refinement could also be obtained with the following sequence of smaller steps:

  $$C \sqcap \forall R.C_1 \overset{[Lit]}{\rightsquigarrow} C \sqcap \forall R.C_1 \sqcap \forall(R \sqcap \ldots).L \overset{[\forall R]}{\rightsquigarrow} \ldots \overset{[\forall R]}{\rightsquigarrow} C \sqcap \forall R.C_1 \sqcap \forall R.L \overset{[\forall\forall]}{=} C \sqcap \forall R.(C_1 \sqcap L).$$

  (In the last step, we applied the simplification rule $[\forall\forall]$.)

- Thirdly, the relationship

$$\exists R_1.C \sqcap \exists R_2.C \sqsupseteq \exists (R_1 \sqcap R_2).C \tag{3}$$

suggests that, for reasons of minimality, we should also drop the $[\exists R]$ altogether. The rule $[\exists R]$ of ρ_0 is redundant since the single step

$$C \sqcap \exists R_1.D \overset{[\exists R]}{\rightsquigarrow} C \sqcap \exists (R_1 \sqcap P).D$$

can also be obtained in several steps, as follows:

$$C \sqcap \exists R_1.D \overset{[Lit]}{\rightsquigarrow} C \sqcap \exists R_1.D \sqcap \exists P.\top \overset{[\exists \top]}{\rightsquigarrow} \ldots \rightsquigarrow C \sqcap \exists R_1.D \sqcap \exists P.D \overset{[\exists\exists]}{\rightsquigarrow} C \sqcap \exists (R_1 \sqcap P).D.$$

Thus, instead of directly refining $\exists R_1.D$ to $\exists (R_1 \sqcap P).D$, we first add $\exists P.D$ and then merge $\exists R_1.D$ and $\exists P.D$ to $\exists (R_1 \sqcap P).D$ using $[\exists\exists]$ (the refinement step being justified by (3)).

- Finally, since the $[Lit]$ step of ρ_0 can add literals L that are complementary to an already existing literal from C, we can obtain inconsistent concepts as refinements of any C. Of course, although $C \rightsquigarrow \bot$ is a valid refinement step (because $C \sqsupseteq \bot$), it is not only non-minimal, but also useless if it is applied on the "top level"[11] of the concept to be refined. However, refining some *subconcept* (of the concept to be refined) to $\bot$ makes sense, for example $\forall R.C \rightsquigarrow \forall R.\bot$ ($\forall R.\bot$ being consistent!), although it is still non-minimal. Unfortunately, as we show below, all refinements $C_1 \rightsquigarrow C_2$ that introduce a new $\bot$ in (some subconcept of) C_2 are always non-minimal, so we have to make a *trade-off between the minimality and the completeness* of the refinement operator. If we want to preserve completeness, we need to allow refinement steps like $C \rightsquigarrow \bot$, if not on the "top level" of the concept to be refined (i.e. in the rule $[Lit]$) or in $[\exists C]$ (where allowing $C \sqcap \exists R_1.C \rightsquigarrow C \sqcap \exists R_1.\bot = \bot$ would lead to an inconsistency), then at least in $[\forall C]$, which has to be modified as follows:
  $[\forall C]$ $\quad C \sqcap \forall R.C_1 \rightsquigarrow C \sqcap \forall R.C_2$ where $C_1 \rightsquigarrow C_2$ **or** $C_2 = \bot$.
  However, besides this explicit introduction of $\bot$ in $\forall$ restrictions, we shall require refinements to be *consistent*. More formally, $D \in \rho(C)$ iff D is obtained as a refinement of C using the refinement rules below ($C \rightsquigarrow D$) and D is *consistent*.

The resulting refinement operator ρ presented below treats literal additions $[Lit]$ in a special manner (since $[Lit]$ is not allowed to be recursively used in $[\exists C]$ or $[\forall C]$ rules). We therefore let ρ' denote all the rules of ρ except $[Lit]$:

[11] i.e. to C, as opposed to its subconcepts.

Refinement rules of ρ'

$[\exists\top]$ $C \sqcap \exists R.\top \rightsquigarrow C \sqcap \exists R.L$ with L a *DL-literal*

$[\exists C]$ $C \sqcap \exists R.C_1 \rightsquigarrow C \sqcap \exists R.C_2$ if $C_1 \overset{\rho'}{\rightsquigarrow} C_2$

$[\exists\exists]$ $C \sqcap \exists R_1.C_1 \sqcap \exists R_2.C_2 \rightsquigarrow C \sqcap \exists(R_1 \sqcap R_2).(C_1 \sqcap C_2)$

$[\forall C]$ $C \sqcap \forall R.C_1 \rightsquigarrow C \sqcap \forall R.C_2$ if $C_1 \overset{\rho'}{\rightsquigarrow} C_2$ or $C_2 = \bot$

$[\forall R]$ $C \sqcap \forall R_1.D \rightsquigarrow C \sqcap \forall R_2.D$ if $R_2 \rightsquigarrow R_1$

$[PR]$ $R \rightsquigarrow R \sqcap P$ with P a primitive role.

In the following, we shall write $C \overset{\textbf{Rule}}{\rightsquigarrow} D$ whenever D is obtained as a refinement of C using the refinement rule **Rule** (which can be $[\exists\top]$, $[\exists C]$, $[\exists\exists]$, $[\forall C]$, $[\forall R]$, or – in the case of ρ – also $[Lit]$). Moreover, we sometimes write $C \overset{\rho'}{\rightsquigarrow} D$ instead of $D \in \rho'(C)$ (denoting the fact that D is obtained as a refinement of C *without using the* $[Lit]$ *rule*).

The refinement rules of the complete refinement operator ρ are the refinement rules of ρ' together with the $[Lit]$ rule.

Refinement rules of ρ

$[\rho']$ refinement rules of ρ'

$[Lit]$ $C \rightsquigarrow C \sqcap L$ with L a *DL-literal* such that $C \sqcap L$ is consistent.

We recall that we have defined $D \in \rho(C)$ iff $C \rightsquigarrow D$ and D is consistent.

3.2 Properties of ρ

1. **Completeness.** Since ρ_0 is complete and the modification of ρ_0 to ρ preserves completeness, ρ will be *complete* too.
   Note that we are adding DL-literals either on the "top level" of the concept to be refined (using $[Lit]$), or inside $\exists R.\top$ restrictions (in $[\exists\top]$). Such literals can be "moved inside" $\forall \cdot \exists \cdot \forall \cdots$ chains by using the $[\exists\exists]$ refinement rule and the $[\forall\forall]$ rewrite rule.
2. **Properness.** Like in the case of ρ_0, ρ is *not proper* because the DL-literals of the form $\exists P.\top$ inserted can be redundant. For achieving properness, we should consider the closure ρ^{cl}, rather than simplifying these redundancies (which would affect completeness).
3. **Minimality.** By construction, ρ has less non-minimal steps than ρ_0. However, there exists a fundamental trade-off between *minimality* and *completeness* of $\mathcal{ALER}$ refinement operators in general (not just for ours).

   Proposition 3. *There exist no* minimal *and* complete $\mathcal{ALER}$ *refinement operators.*[12]

[12] There cannot exist a *minimal* refinement step $C \rightsquigarrow \forall P.\bot$, since there exists a sufficiently large n (for example, larger than the size of C) such that $C \sqsupset C \sqcap \underbrace{\forall P. \ldots \forall P.}_{n} A \sqsupset \forall P.\bot$.

Example 2. The following infinite chain of minimal refinements between $\forall P.A$ and $\forall P.\bot$

$$\begin{aligned} C &= \forall P.A \ \sqsupset \\ C_1 &= \forall P.(A \sqcap \forall P.A) \ \sqsupset \\ C_2 &= \forall P.(A \sqcap \forall P.(A \sqcap \forall P.A)) \ \sqsupset \\ C_3 &= \forall P.(A \sqcap \forall P.(A \sqcap \forall P.(A \sqcap \forall P.A))) \ \sqsupset \ldots \sqsupset \\ C_\infty &= \forall P.\bot \end{aligned}$$

shows that a *minimal* refinement operator will not be able to reach $\forall R.\bot$ from $\forall P.A$ in a *finite* number of steps, thereby being *incomplete*.
On the other hand, if we insist on *completeness*, we should allow $\forall P.\bot$ as a refinement of some C_i, thereby making the refinement operator *non-minimal*.

Our refinement operator allows $\forall P.\bot$ as a refinement of any C_i in the example above. It is therefore non-minimal. However, we conjecture that although there exist no minimal and complete refinement operators for $\mathcal{ALER}$, there exist refinement operators all of whose steps $C \rightsquigarrow D$ are minimal, *except for the steps involving the introduction of* $\bot$ *in some subconcept of* D (like in the $C_2 = \bot$ case of the $[\forall C]$ rule of our refinement operator ρ). This suggests that our ρ is one of the best refinement operators one can hope for.

4. **Local finiteness.** The following example shows that there can be no locally finite and complete $\mathcal{ALER}$ refinement operators.

   Example 3. A_1 admits the following infinite set of minimal direct refinements:

   $$\{A_1 \sqcap \forall P.A, \quad A_1 \sqcap \forall P.\forall P.A, \quad A_1 \sqcap \forall P.\forall P.\forall P.A, \quad \ldots\}.$$

   Therefore, since ρ is complete, it will not be locally finite either. Apparently, this seems to be a significant problem. However, as the example above suggests, the infinite set of minimal direct refinements of some concept C involves increasingly longer concepts D, which will be immediately discarded by a refinement heuristic taking into account the size of hypotheses:

   $$f(H) = pos(H) - neg(H) - size(H)$$

   (where $pos(H)$ and $neg(H)$ are the number of positive/negative examples covered by the hypothesis H.)
   Note that the lack of local finiteness and respectively minimality of a complete $\mathcal{ALER}$ refinement operator seem to be related. (They seem to involve value restrictions and $\bot$ in an essential way[13]). This situation also seems

[13] The situation is unlike in ILP, where such problems do not occur (obviously, because value restrictions cannot be expressed in Logic Programming).

In $\mathcal{ALER}$, on the other hand, we do not have infinite ascending chains [18] because $\mathcal{ALER}$ does not allow the construction of cyclic descriptions (such as $R(X_1, X_2), R(X_2, X_3), R(X_3, X_1)$, for example).

to be related to the non-existence of the Most Specific Concept (MSC) of individuals in some concept languages, such as $\mathcal{ALN}$ [1].

4 Testing Example Coverage

Although both Horn-clause logic programming (LP) and description logics (DL) are fragments of first order logic and are therefore similar in certain respects, there are also some significant differences.

- DLs make the *Open World Assumption (OWA)*, while LP makes the *Closed World Assumption (CWA)*.
- DL definitions like $A \leftarrow \forall R.C$ and $A \rightarrow \exists R.C$ involve existentially quantified variables and thus cannot be expressed in pure LP. Using the meta-predicate *forall*, we could approximate $A \leftarrow \forall R.C$ as $A(X) \leftarrow forall(R(X,Y), C(Y))$. But while the former definition is interpreted w.r.t. the OWA, the latter is interpreted w.r.t. the CWA, which makes it closer to $A \leftarrow \forall KR.C$. where K is an epistemic operator as in [11]. Also, while DLs provide inference services (like subsumption checking) to reason about such descriptions, LP systems with the meta-predicate *forall* can only answer queries, but not reason about such descriptions.

Although the OWA is sometimes preferable to the CWA[14], the OWA is a problem when testing that a definition, for example $A \leftarrow \forall R.C$, covers a given example, for instance $A(a_i)$. Examples are unavoidably *incomplete*. Even if all the *known* R-fillers of a_i from the Abox verify C:

$$\mathcal{A} = \{R(a_i, b_{i1}), C(b_{i1}), \ldots, R(a_i, b_{in}), C(b_{in})\}$$

this doesn't mean that a_i verifies $\forall R.C$[15], so the antecedent $\forall R.C$ of $A \leftarrow \forall R.C$ will never be satisfied by an example a_i (unless explicitly stated in the KB). However, a_i will verify $\forall KR.C$ because all the *known* R-fillers of a_i verify C, so the definition $A \leftarrow \forall KR.C$ covers the example $A(a_i)$, as expected.

Thus, when checking example coverage. we need to "close" the roles (for example, by replacing R with KR, or, equivalently, assuming that the known fillers are all the fillers).

4.1 Example Coverage for Sufficient Definitions

Definition 6. *In the case of a DL knowledge base* $\langle \mathcal{T}, \mathcal{A} \rangle$*, for which* $\mathcal{A}' = \{A(a_1), \ldots, A(a_p), \neg A(a_{p+1}), \ldots, \neg A(a_{p+n})\} \subseteq \mathcal{A}$ *are considered (positive and negative) examples, we say that the sufficient definition* $A \leftarrow C$ covers *the (positive or negative) example* a_i *iff*

$$cl\langle \mathcal{T} \cup \{A \leftarrow C\}, \mathcal{A} \setminus \mathcal{A}' \rangle \models A(a_i),$$

[14] For example, when expressing constraints on role fillers using value restrictions.

[15] because of the CWA, there could exist, in principle, a yet unknown R-filler of a_i not verifying C.

which is equivalent with the inconsistency of

$$cl\langle \mathcal{T} \cup \{A \leftarrow C\}, (\mathcal{A} \setminus \mathcal{A}') \cup \{\neg A(a_i)\}\rangle,$$

where $cl\langle \mathcal{T}, \mathcal{A}\rangle$ *denotes the role-closure of the knowledge base* $\langle \mathcal{T}, \mathcal{A}\rangle$ *(which amounts to replacing roles* R *with* KR*).*

Example 4. Consider the following knowledge base $\langle \mathcal{T}, \mathcal{A}\rangle$ with an empty Tbox ($\mathcal{T} = \emptyset$) and the Abox[16]

$$\begin{aligned}\mathcal{A} = \{&IP(j), R(j), F(j, j_1), I(j_1), F(j, j_2), I(j_2),\\ &\neg IP(f), R(f),\\ &\neg IP(m), R(m), F(m, m_1), I(m_1), F(m, m_2),\\ &\neg IP(h), F(h, h_1), I(h_1), F(h, h_2), I(h_2)\},\end{aligned}$$

where $\mathcal{A}' = \{IP(j), \neg IP(f), \neg IP(m), \neg IP(h)\}$ are considered as positive and negative examples.

The following sufficient definition covers all positive examples while avoiding all (not covering any) negative examples:

$$IP \leftarrow R \sqcap \forall F.I \sqcap \exists F.I \tag{4}$$

(4) covers the positive example $IP(j)$ because $\langle \{IP \leftarrow R \sqcap \forall KF.I \sqcap \exists KF.I\}, (\mathcal{A} \setminus \mathcal{A}') \cup \{\neg IP(j)\}\rangle$ is inconsistent, since $\neg R(j)$, $(\neg\forall KF.I)(j)$, $(\neg\exists KF.I)(j)$ are all inconsistent:

- $\neg R(j)$ because $R(j) \in \mathcal{A} \setminus \mathcal{A}'$
- $(\neg\forall KF.I)(j)$, i.e. $(\exists KF.\neg I)(j)$ because all the *known* F-fillers of j (namely j_1 and j_2) are I
- $(\neg\exists KF.I)(j)$, i.e. $(\forall KF.\neg I)(j)$ because there exists an F-filler of j (for example j_1) that is I.

Note that the more general definition $IP \leftarrow R \sqcap \forall F.I$ covers the negative example $\neg IP(f)$ because $\langle \{IP \leftarrow R \sqcap \forall KF.I\}, (\mathcal{A} \setminus \mathcal{A}') \cup \{\neg IP(f)\}\rangle$ is inconsistent, since $\neg R(f)$, $(\neg\forall KF.I)(f)$ are both inconsistent:

- $\neg R(f)$ because $R(f) \in \mathcal{A} \setminus \mathcal{A}'$
- $(\neg\forall KF.I)(f)$, i.e. $(\exists KF.\neg I)(f)$ because there exists no *known* F-filler of f.

And since the more specific (4) does not cover the negative example $\neg IP(f)$ (due to the consistency of $(\neg\exists KF.I)(f)$), we conclude that $\exists F.I$ discriminates between the positive example $IP(j)$ and the negative example $\neg IP(f)$ and is therefore necessary in (4).

$IP \leftarrow \forall F.I \sqcap \exists F.I$ (which is also more general than (4)) covers the negative example $\neg IP(h)$ because $\langle \{IP \leftarrow \forall KF.I \sqcap \exists KF.I\}, (\mathcal{A} \setminus \mathcal{A}') \cup \{\neg IP(h)\}\rangle$ is inconsistent, since $(\neg\forall KF.I)(h)$, $(\neg\exists KF.I)(h)$ are inconsistent:

[16] Where the individuals *j, m, f, h* stand for *John, Mary, Fred, Helen*, while the atomic concepts *IP, R, I* stand for *Influencial_Person, Rich, Influencial* and the primitive role F for *Friend*.

- $(\neg\forall KF.I)(h)$, i.e. $(\exists KF.\neg I)(h)$ because all the *known* F-fillers of h (namely h_1 and h_2) are I
- $(\neg\exists KF.I)(h)$, i.e. $(\forall KF.\neg I)(h)$ because there exists an F-filler of h (for example h_1) that is I.

And since the more specific (4) does not cover the negative example $\neg IP(h)$ (due to the consistency of $\neg R(h)$), we conclude that R discriminates between the positive example $IP(j)$ and the negative example $\neg IP(h)$ and is therefore necessary in (4).

$IP \leftarrow R \sqcap \exists F.I$ (which is also more general than (4)) covers the negative example $\neg IP(m)$ because $\langle\{IP \leftarrow R \sqcap \exists KF.I\}, (\mathcal{A} \setminus \mathcal{A}') \cup \{\neg IP(m)\}\rangle$ is inconsistent, since $\neg R(m)$, $(\neg\exists KF.I)(m)$ are inconsistent:

- $\neg R(m)$ because $R(m) \in \mathcal{A} \setminus \mathcal{A}'$
- $(\neg\exists KF.I)(m)$, i.e. $(\forall KF.\neg I)(m)$ because there exists an F-filler of m (namely m_1) that is I.

And since the more specific (4) does not cover the negative example $\neg IP(m)$ (due to the consistency of $(\neg\forall KF.I)(m)$), we conclude that $\forall F.I$ discriminates between the positive example $IP(j)$ and the negative example $\neg IP(m)$ and is therefore necessary in (4).

The right-hand side of (4) is obtained by the refinement operator ρ from $\top$ by the following sequence of steps:
$C_0 = \top \overset{[Lit]}{\rightsquigarrow} C_1 = R \overset{[Lit]}{\rightsquigarrow} C_2 = R \sqcap \forall F.I \overset{[Lit]}{\rightsquigarrow} C_3 = R \sqcap \forall F.I \sqcap \exists F.\top \overset{[\exists\forall]}{=} R \sqcap \forall F.I \sqcap \exists F.I$.

All C_i above cover the positive example $IP(j)$. However, C_0 covers all 3 negative examples, C_1 only $\neg IP(f)$, $\neg IP(m)$, C_2 only $\neg IP(f)$, while C_3 avoids all negative examples and would be returned as a solution: $IP \leftarrow C_3$.

4.2 Verifying Necessary Definitions

While sufficient definitions can be used to classify individuals as instances of the target concept, necessary definitions impose constraints on the instances of the target concept. Roughly speaking, a necessary definition $A \rightarrow C$ is *verified* iff it is entailed by the knowledge base: $\langle\mathcal{T}, \mathcal{A}\rangle \models (A \rightarrow C)$, which can be reduced to the inconsistency of $A \sqcap \neg C$ w.r.t. $\langle\mathcal{T}, \mathcal{A}\rangle$, i.e. to the non-existence of an instance x of $A \sqcap \neg C$. Such an x could be either a_i[17], another *known* individual, or a *new* one. Since the examples a_i of A are unavoidably incomplete, $A \rightarrow C$ will not be *provable* for all imaginable instances x. Equivalently, $(A \sqcap \neg C)(x)$ will not (and need not) be inconsistent for any x. In fact, we need to prove $A \rightarrow C$ (or, equivalently, to check the inconsistency of $A \sqcap \neg C$) only for the *known* examples a_i of A. This amounts to proving $(KA) \rightarrow C$, i.e. to considering the closure of the target concept A. Since $A(a_i)$ holds anyway, we just need to prove $C(a_i)$ for all positive examples a_i of A. More precisely:

[17] Assuming that the known positive examples of A are $\mathcal{A}' = \{A(a_1), \ldots, A(a_n)\}$.

Definition 7. *The necessary definition* $A \rightarrow C$ *is* verified *in* $\langle \mathcal{T}, \mathcal{A} \rangle$ *iff* $\forall A(a_i) \in \mathcal{A}'$, $cl\langle \mathcal{T}, \mathcal{A} \cup \{\neg C(a_i)\} \rangle$ *is inconsistent.*

Note that the above definition can be considered to obey the so-called *'provability view'* of constraints. (Adopting the weaker *'consistency view'* may be too weak in our learning framework: assuming that a constraint is verified just because it is consistent with the examples may lead to the adoption of too strong – and thus unjustified – constraints.)

5 Learning in DLs Using Refinement Operators

A top-down DL learning algorithm would simply refine a very general definition of the target concept, like $A \leftarrow \top$, using a downward refinement operator[18] until it covers no negative examples. (A heuristic maximizing the number of positive examples covered, while minimizing the size of the definitions as well as the number of negative examples covered can be used to guide the search.) If this first covering step still leaves some positive examples uncovered, then subsequent covering steps will be employed to learn additional definitions until either all positive examples are covered, or the learning process fails due to the impossibility to cover certain positive examples without also covering (some) negative examples.

Note that our approach avoids the difficulties faced by bottom-up approaches, which need to compute the minimal Tbox generalizations $MSC(a_i)$ (called *most specific concepts* [8,11]) of the Abox examples $A(a_i)$. Most existing bottom-up approaches (such as [8]) then use *least common subsumers (LCS)* [7,8,2] to generalize the MSC descriptions. Unfortunately, such approaches tend to produce overly specific concept definitions. On the other hand, by reverting the arrows in our downward refinement operator, we obtain an *upward* refinement operator for the description logic $\mathcal{ALER}$ which can be used to search the space of DL descriptions in a more flexible way than by using LCSs and also without being limited to considering only least generalizations.

6 Conclusions

This paper can be viewed as an attempt to apply ILP learning methods to description logics – a widely used knowledge representation formalism that is different from the language of Horn clauses, traditionally employed in ILP. Extending ILP learning methods to description logics is important for at least two reasons. First, description logics represent a new sort of *learning bias*, which necessitates a more sophisticated refinement operator (than typical ILP refinement operators). Second, description logics provide constructs, such as value

[18] The inherent redundancy of the *complete* refinement operator ρ needs to be eliminated by converting it into a *weakly complete* operator. This can be done using the methods from [3,4].

restrictions, which cannot be expressed in Horn logic, thereby enhancing the expressivity of the language and making it more suitable for applications that involve rich hierarchical knowledge.

Since Horn logic (HL) and description logics (DLs) are complementary, developing refinement operators for DLs represents a significant step towards learning in an integrated framework comprising both HL and DL. Due to the differences in expressivities between DLs and HL, constructing DL and respectively HL refinement operators encounter different problems. Neither can be minimal (while preserving completeness), but for different reasons: in HL we can have infinite ascending chains, while in our $\mathcal{ALER}$ description logic we cannot, non-minimality being due to the interplay of value restrictions and $\bot$. We also discuss the impact of the Open World Assumption (usually employed in DLs) on learning and especially on example coverage, but this issue needs further investigation.

Since learning in even a very simple DL allowing for definitions of the form $C \leftarrow \exists R.(A_1 \sqcap \ldots \sqcap A_n)$ is NP-hard (Theorem 2 of [13]), learning in our framework will be NP-hard as well. However, we prefer to preserve a certain expressiveness of the language and plan to study more deeply the *average case* tractability of our approach (for which *worst-case* intractability is less relevant).

We are currently exploring methods of taking into account Tbox definitions $\mathcal{T}$ as background knowledge during refinement, as we plan to further reduce the non-minimality of our refinement operator w.r.t. $\mathcal{T} \neq \emptyset$ (a refinement step that is minimal w.r.t. $\mathcal{T} = \emptyset$ could be non-minimal when taking into account the definitions of some non-empty $\mathcal{T}$). Note that completeness is not an issue in this case, since a refinement operator that is complete w.r.t. $\mathcal{T} = \emptyset$ will remain complete w.r.t. a non-empty terminology.

Acknowledgments. The first author is grateful to Doina Ţilivea for discussions. Thanks are also due to the anonymous reviewers for their suggestions and constructive criticism.

References

1. Baader F., Küsters R. *Least common subsumer computation w.r.t. cyclic ALN-terminologies.* In Proc. Int. Workshop on Description Logics (DL'98), Trento, Italy.
2. Baader F., R. Küsters, R. Molitor. *Computing Least Common Subsumers in Description Logics with Existential Restrictions.* Proc. IJCAI'99, pp. 96-101.
3. Badea Liviu, Stanciu Monica. *Refinement Operators Can Be (Weakly) Perfect.* Proc. ILP-99, LNAI 1631, Springer, 1999, pp. 21-32.
4. Badea Liviu. *Perfect Refinement Operators Can Be Flexible.* Proc. ECAI-2000.
5. Borgida A. *On the relative Expressiveness of Description Logics and Predicate Logics.* Artificial Intelligence, Vol. 82, Number 1-2, pp. 353-367, 1996.
6. Buchheit M., F. Donini, A. Schaerf. *Decidable reasoning in terminological knowledge representation systems.* J. Artificial Intelligence Research, 1:109-138, 1993.
7. Cohen W.W., A. Borgida, H. Hirsh. *Computing least common subsumers in description logics.* Proc. AAAI-92, San Jose, California, 1992.
8. Cohen W.W., H. Hirsh. *Learning the CLASSIC description logic: Theoretical and experimental results.* In Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference, pp. 121-133, 1994.

9. Donini F.M., M. Lenzerini, D. Nardi, W. Nutt. *The complexity of concept languages.* Information and Computation, 134:1-58, 1997.
10. Donini F.M., M. Lenzerini, D. Nardi, A. Schaerf *AL-log: integrating datalog and description logics.* Journal of Intelligent Information Systems, 10:227-252, 1998.
11. Donini F.M., M. Lenzerini, D. Nardi, A. Schaerf, W. Nutt. *An epistemic operator for description logics.* Artificial Intelligence, 100 (1-2), 225-274, 1998.
12. Kietz J.U., Morik K. *A Polynomial Approach to the Constructive Induction of Structural Knowledge.* Machine Learning, Vol. 14, pp. 193-217, 1994.
13. Kietz J.U. *Some lower-bounds for the computational complexity of Inductive Logic Programming.* Proc. ECML'93, LNAI 667, Springer, 1993.
14. Levy A., M.C. Rousset. *CARIN: A Representation Language Combining Horn Rules and Description Logics.* Proc. ECAI-96, Budapest, 1996.
15. Levy A., M.C. Rousset. *The Limits on Combining Horn Rules with Description Logics.* Proc. AAAI-96, Portland, 1996.
16. Muggleton S. *Inverse entailment and Progol.* New Generation Computing Journal, 13:245-286, 1995.
17. van der Laag P., S.H. Nienhuys-Cheng. *A Note on Ideal Refinement Operators in Inductive Logic Programming.* Proceedings ILP-94, 247-260.
18. van der Laag P., S.H. Nienhuys-Cheng. *Existence and Nonexistence of Complete Refinement Operators.* ECML-94, 307-322.
19. Nienhuys-Cheng S.H., de Wolf R. *Foundations of Inductive Logic Programming.* LNAI 1228, Springer Verlag, 1997.
20. Nienhuys-Cheng S.-H., W. Van Laer, J. Ramon, and L. De Raedt. *Generalizing Refinement Operators to Learn Prenex Conjunctive Normal Forms.* Proc. ILP-99, LNAI 1631, Springer, 1999, pp. 245-256.

Executing Query Packs in ILP

Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Hendrik.Blockeel,Luc.Dehaspe,Bart.Demoen,Gerda.Janssens,
Jan.Ramon,Henk.Vandecasteele}@cs.kuleuven.ac.be

Abstract. Inductive logic programming systems usually send large numbers of queries to a database. The lattice structure from which these queries are typically selected causes many of these queries to be highly similar. As a consequence, independent execution of all queries may involve a lot of redundant computation. We propose a mechanism for executing a hierarchically structured set of queries (a "query pack") through which a lot of redundancy in the computation is removed. We have incorporated our query pack execution mechanism in the ILP systems TILDE and WARMR by implementing a new Prolog engine ILPROLOG which provides support for pack execution at a lower level. Experimental results demonstrate significant efficiency gains. Our query pack execution mechanism is very general in nature and could be incorporated in most other ILP systems, with similar efficiency improvements to be expected.

1 Introduction

Many data mining algorithms, including ILP algorithms, employ an approach that is basically a generate-and-test approach: large numbers of hypotheses are generated and tested against the database in order to check whether they are valid. Even though their search through a hypothesis space is seldom exhaustive in practical situations, and clever branch-and-bound or greedy search strategies are employed, the number of hypotheses generated and tested by these approaches may still be huge. This is especially true when a complex hypothesis space is used, which is often the case in ILP.

Very often the hypothesis space is structured as a lattice, and the search through the space makes use of this lattice. Because hypotheses close to one another in the lattice are similar, the computations involved in testing their validity will be similar too. In other words, many of the computations that are performed when executing one query will have to be performed again when executing the next query. Storing certain intermediate results during the computation for later use could be a solution (e.g., tabling as in the XSB Prolog engine [7]), but may be infeasible in practice because of its memory requirements. It becomes more feasible if the search is reorganised so that intermediate results are always used shortly after they have been computed; this can be achieved to some extent by

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 60–77, 2000.

rearranging the computations. The best way of removing the redundancy, however, is to re-implement the execution strategy of the queries so that as much computation as possible is effectively shared.

In this paper we discuss a strategy for executing sets of queries, organised in so-called query packs, that avoids these redundant computations. The strategy is presented as an adaptation of the standard Prolog execution mechanism. We also report on the use of this technique by several inductive logic programming systems. Experimental results suggest that in some cases a speed-up of an order of magnitude or more can be achieved in this way. This significantly contributes to the applicability of inductive logic programming to real world data mining tasks.

The remainder of this paper is structured as follows. In Section 2 we precisely describe the ILP problem setting in which this work is set. In Section 3 we define the notion of a query pack, indicate how it could be executed by a standard Prolog interpreter and what computational redundancy this causes, and propose an algorithm to execute such query packs that avoids these redundant computations. In Section 4 we describe how the query pack execution strategy can be incorporated in two existing inductive logic programming algorithms (TILDE and WARMR). In Section 5 we present experimental results that give an indication of the speed-up that these systems achieve by using the query pack execution mechanism. In Section 6 related work is mentioned and in Section 7 we conclude.

2 Problem Setting

The problem setting we are considering is the following [8]:

Given

- a set of conjunctive queries S
- a deductive database D
- a tuple K of variables that occur in all queries in S

Find

- the set $R = \{(K\theta, Q) | Q \in S \text{ and } Q\theta \text{ succeeds in } D\}$; i.e., find for each query Q in S those instantiations of K for which the query succeeds in D.

We refer to the tuple K as a *key*; the intuition behind K is that it uniquely identifies a single example. In the context of learning a single predicate the key would typically consist of the variables that occur in the head of the clause being learned.

Example 1. Assume an ILP system learning a definition for `father/2` wants to evaluate the following hypotheses:

```
father(X,Y) :- parent(X,Y), male(X).
father(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(Y).
father(X,Y) :- parent(X,Y), female(Y).
```

Examples are of the form `father(`c_1`,`c_2`)` with c_1 and c_2 some constants; hence in the above set of clauses each example is uniquely identified by a ground substitution of the tuple (X, Y). This means that, put in the above problem setting, we have a set of Prolog queries S = $\{(parent(X,Y), male(X)), (parent(X,Y), female(X)), (parent(X,Y), male(Y)), (parent(X,Y), female(Y))\}$ and a key $K = (X, Y)$. Given a query $Q \in S$, finding all couples (x, y) for which $((x, y), Q) \in R$ (with R the result set as defined above) is equivalent to finding all `parent(`c_1`,`c_2`)` facts predicted by the clause `parent(X,Y) :-` Q.

Our problem setting is very general in the sense that it covers many tasks typically encountered in ILP settings. Indeed, once it is known which queries succeed for which examples, statistics about the queries can be readily obtained from this. A few examples:

- discovery of frequent item-sets: for each query Q the number of keys for which it succeeds just needs to be counted, i.e., $|\{K\theta | (K\theta, Q) \in R\}|$
- induction of Horn clauses: the accuracy of a clause $H :- B$ can be computed as $\frac{|\{K\theta | (K\theta,B) \in R \text{ and } D \models H\theta\}|}{|\{K\theta | (K\theta,B) \in R\}|}$ with R the result set
- induction of classification or regression trees: computing the class entropy or variance of the examples covered (or not) by a query involves simple computations on counts similar to the above ones

Obviously our problem setting returns more information than most systems need; in practice one could avoid computing the result set itself and just update some relevant counters instead. By considering the above setting any efficiency results we obtain are *a fortiori* applicable to systems avoiding the computation of this overhead of information.

3 Query Packs

3.1 Definition

Definition 1. *A query pack is a tree structure with literals or conjunctions of literals in the nodes. Each path from the root to some node (leaf or internal node) represents a conjunctive query. Such a query is said to be a* member *of the query pack.*

Example 2. Given the following set of queries:

```
p(X)
p(X), q(X,a)
p(X), q(X,b)
p(X), q(X,c)
p(X), q(X,Y)
p(X), q(X,Y), r(Y,1)
p(X), q(X,Y), r(Y,2)
```

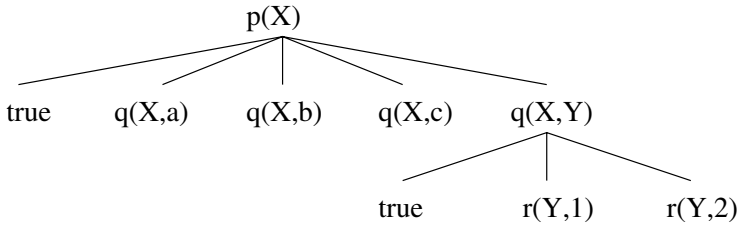


Fig. 1. A query pack

the query pack shown in Fig. 1 contains these and only these queries as members. We represent the pack as a Prolog term as

p(X), (true **or** q(X,a) **or** q(X,b) **or** q(X,c) **or** q(X,Y), (true **or** r(Y,1) **or** r(Y,2)))

As in the example, we will denote query packs textually by writing an **or** operator between different subtrees of a node. We name the operator **or** because intuitively it closely corresponds to a disjunction. Indeed a query pack could be constructed and executed as a complex Prolog query reflecting the tree structure of the pack (with a ";" separating children of the same node and a "," separating a node from its children): the set of answer substitutions for this query is then exactly the union of the sets of answer substitutions for the members of the pack. Executing the pack in this manner would indeed remove a lot of redundant computation, but is still suboptimal, as the following example shows.

Example 3. Consider the following set of queries, with X as key:

```
p(X), lotsofwork(X), q(X,Y)
p(X), lotsofwork(X), r(X,Y)
```

where `lotsofwork` represents a complex computation. Evaluating the queries separately implies that for each X for which `p(X)` succeeds, `lotsofwork(X)` will be computed twice, whereas for the following query

```
p(X), lotsofwork(X), (q(X,Y); r(X,Y))
```

`lotsofwork(X)` will only be called once for each X. Both queries are equivalent with respect to their answer substitutions.

On the other hand, consider the following query:

```
p(X), q(X,Y), (r(Y,Z); s(Y,Z))
```

Suppose `r(Y,Z)` associates many Z's with each single Y. With the above query, all alternative values for Z will be generated by `r(Y,Z)` before `s(Y,Z)` be called. Given that we are only interested in answer substitutions for X, getting one single answer substitution for Z suffices and the alternatives for `r(Y,Z)` should be cut away. Note that the cut should survive backtracking to some extent: when backtracking to `q(X,Y)` yields a new value for Y, the `r(Y,Z)` branch should still be avoided because we already know that it succeeds for this value of X. Only when the value of the key X changes should it be reconsidered.

The execution mechanism we need can be implemented in Prolog, but only in a very inefficient manner [11, 2]; actually preliminary results in [2] suggested that the overhead involved in this implementation destroys the efficiency gain obtained by redundancy reduction. To fully exploit the sharing of computations changes are needed at the level of the Prolog engine itself.

3.2 Efficient Execution of Query Packs

Several possible solutions for efficient execution of query packs are described in [11]. The most efficient one consists of an extension of the WAM (Warren Abstract Machine), the machine underlying most Prolog implementations. The extended WAM provides the **or** operator as discussed above, and permanently removes branches from the pack that do not need to be investigated anymore. This extended WAM is the basis of a new Prolog engine dedicated to inductive logic programming, called ILPROLOG.

The starting point for the query pack execution mechanism is the usual Prolog execution of a query Q given a Prolog program P. By backtracking Prolog will generate all the solutions for Q by giving the possible instantiations θ such that $Q\theta$ succeeds in P.

In this context, a query pack consists of a conjunction $conj$ of literals followed by a set of branches, where each branch is again a query pack. Note that leaves are query packs with an empty set of branches. For each query pack $\mathcal{Q}$, $conj(\mathcal{Q})$ denotes the conjunction and $children(\mathcal{Q})$ denotes the set of branches.

A (root) query pack actually represents a set of queries. Execution of a query pack aims at finding out which queries of the pack succeed. If a query pack is executed as if the **or**'s were usual disjunctions, backtracking occurs over queries that have already succeeded and too many successes are detected. To avoid this, some part of the query pack should no longer be considered during backtracking as soon as a query succeeds. The algorithm realises this by reporting success of queries (and of query packs) to points higher up in the query pack.

A (non-root) query pack can be safely removed if all the queries that depend on it (namely all the queries that are below it in the query pack) have succeeded once. For a leaf $\mathcal{Q}$ (empty set of children), success of $conj(\mathcal{Q})$ is sufficient to remove it. For a $\mathcal{Q}$ with dependent queries, we wait until all the dependent queries report success or equivalently until all the query packs in $children(\mathcal{Q})$ report success.

At the start of the evaluation of a root query pack, the set of branches for every query pack in it contains all the branches in the given query pack. During the execution, query packs can be removed from sets and the values of the $children(\mathcal{Q})$ changes accordingly. Thus, when due to backtracking a query pack is executed again, it might be the case that fewer branches have to be considered.

The execution of a query pack $\mathcal{Q}\theta$ is defined by the algorithm $execute\_qp(\mathcal{Q}, \theta)$ (Fig. 2) which imposes additional control on the usual Prolog execution.

The usual Prolog execution and backtracking behaviour is modelled by the while loop (line 1) which generates all possible solutions σ for the conjunction in

```
0   execute_qp( pack Q, substitution θ) {
1   while ( σ ← next_solution( conj(Q)θ)
2       {
3         for each Q_child in children(Q) do
4             {
5               if ( execute_qp( Q_child , σ) == success)
6                   children(Q) ← children(Q) \ {Q_child}
7             }
8         if ( children(Q) is an empty set) return(success)
9       }
10  return(fail)
11  }
```

Fig. 2. The query pack execution algorithm

the query pack. If no more solutions are found, fail is returned and backtracking will occur at the level of the calling query pack.

The additional control manages the $children(\mathcal{Q})$. For each solution σ, the necessary branches of $\mathcal{Q}$ will be executed. It is important to notice that the initial set of branches of a query pack is changed destructively during the execution of this algorithm. Firstly, when a leaf is reached, success is returned (line 8) and the corresponding branch is removed from the query pack (line 6). Secondly, when a query pack that initially had several branches, finally ends up with an empty set of branches (line 6), also this query pack is removed (line 8). The fact that branches are destructively removed, implies that when due to backtracking the same query pack is executed again for a different σ, not all branches have to be executed any more. Moreover, by returning success the backtracking over the current query pack conjunction $conj(\mathcal{Q})$ is stopped: all branches have reported success.

3.3 Using Query Packs

Fig. 3 shows an algorithm that makes use of the pack execution mechanism to compute the result set R as defined in our problem statement. From a query pack $\mathcal{Q}$ containing all queries in S, a derived pack $\mathcal{Q}'$ is constructed by adding a `report_success/2` literal to each leaf of the pack; the task of `report_success(K,N)` is simply to add (K, Q_N) to R (with Q_N the N-th query in the pack).[1] $\mathcal{Q}'$ will be executed via the predicate `evaluate_pack/1` which is dynamically defined, compiled, loaded and then executed. Obviously an ILP sys-

[1] In our current implementation the result set is implemented as a bit-matrix indexed on queries and examples. This implementation is feasible (on typical computers at this moment) as long as the number of queries in the pack multiplied by the number of examples is less than 10^9, which holds for most current ILP applications.

```
1  evaluate(set of examples E, pack Q, key K) {
2      Q' ← Q;
3      q ← 1;
4      for each leaf of Q' do {
5          add report_success(K, q) to the right of the conjunction in the leaf
6          increment q
7      }
8      C ← (evaluate_pack(K) :- Q');
9      compile_and_load(C);
10     for each example e in E do {
11         k ← key(e);
12         evaluate_pack(k);
13     }
14 }
```

Fig. 3. Using query packs to compute the result set R

tem not interested in the result set itself could provide its own `report_success/2` predicate and thus avoid the overhead of explicitly building the result set.

Note that this algorithm follows the strategy of running all queries for each single example before moving on to the next example: this could be called the "examples in outer loop" strategy, as opposed to the "queries in outer loop" strategy used by most ILP systems. The "examples in outer loop" strategy has important advantages when processing large data sets; see, e.g., [12, 5].

3.4 Computational Complexity

Lower and upper bounds on the speedup factor that can be achieved by executing a pack instead of separate queries can be obtained as follows. For a pack containing n queries $q_i = (a, b_i)$, let T_i be the time needed to compute the first answer substitution of q_i if there are any, or to obtain failure otherwise. Let t_i be the part of T_i spent within a and t'_i the part of T_i spent in b_i. Then $T_s = \sum_i (t_i + t'_i)$ and $T_p = \max(t_i) + \sum_i t'_i$ with T_s and T_p representing the total time needed for executing all queries separately, respectively executing the pack. Introducing $c = \sum_i t_i / \sum_i t'_i$, it is possible to show that $T_s/T_p \geq 1$ and $T_s/T_p \leq (c+1)/(c/n+1) < \min(c+1, n)$. Thus the speedup factor is bound by the branching factor n and by the ratio c of computational complexity in the shared part over the computational complexity of the non-shared part. For multi-level packs, under the assumption of a constant branching factor b, if the computation of literals at depth 1 to i in the pack is dominant then the speedup factor approaches b^{d-i} with d the depth of the pack.

4 Use of Query Pack Execution in ILP Systems

In this section we briefly describe two existing ILP algorithms and show how the above execution method can be included in the algorithms to improve efficiency.

4.1 Refinement of a Single Rule

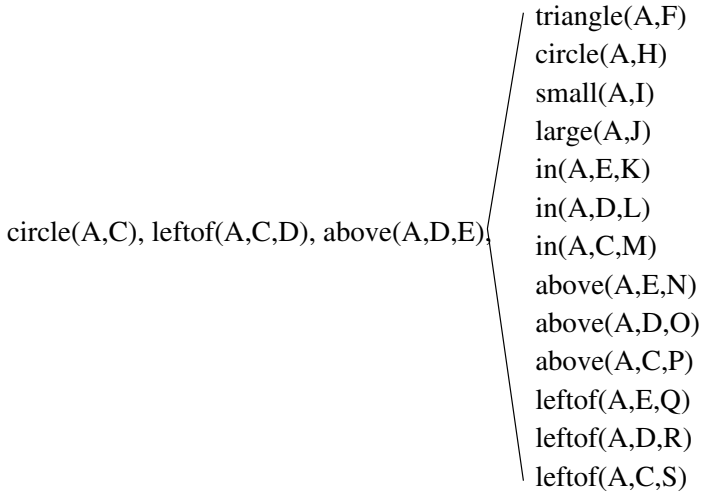
The first algorithm we discuss is TILDE[4], an algorithm that builds first-order decision trees. In a first-order decision tree, nodes contain literals that together with the conjunction of the literals in the nodes above this node (i.e., in a path from the root to this node) form the query that is to be run for an example to decide which subtree it should be sorted into. When building the tree, the literal (or conjunction of literals) to be put in one node is chosen as follows: given the query corresponding to a path from the root to this node, generate all refinements of this query (a refinement of a query is formed by adding one or more literals to the query); evaluate these refinements on the data set (computing, e.g., the information gain [14] yielded by the refinement), choose the best refinement, and put the literals that were added to the original clause to form this refinement in the node.

At this point it is clear that a lot of computational redundancy exists if each refinement is evaluated separately. Indeed all refinements contain exactly the same literals except those added during this single refinement step. Organising all refinements into one query pack, we obtain a query pack that essentially has only one level (the root immediately branches into leaves). When TILDE's lookahead facility is used [3], refinements form a lattice and the query pack may contain multiple (though usually few) levels.

Note that the root of these packs may consist of a conjunction of many literals, giving the pack a broom-like form. The more literals in the root of the pack, the greater the benefit of query pack execution is expected to be.

Example 4. Assume the node currently being refined has the following query associated with it: `circle(A,C),leftof(A,C,D),above(A,D,E)`, i.e., the node covers all examples where there is a circle to the left of some other object which is itself above yet another object.

The query pack generated for this refinement could for instance be



When evaluating this pack, all backtracking through the root of the pack (the "stick" of the broom) will happen only once, instead of once for each refinement. In other words: when evaluating queries one by one, for each query the Prolog engine needs to search once again for all objects C, D and E fulfilling the constraint `circle(A,C), leftof(A,C,D), above(A,D,E)`; when executing a pack this search is only done once.

Other Systems Besides Tilde Many systems for inductive logic programming use an algorithm that consists of repeatedly refining clauses. Any of these systems could in principle be rewritten to make use of a query pack evaluation mechanism and thus achieve a significant efficiency gain. Consider, e.g., a system performing an A^* search through a refinement lattice, such as PROGOL [13]. Since A^* imposes a certain order in which clauses will be considered for refinement, it is hard to reorganise the computation at this level. However, when taking one node in the list of open nodes and producing all its refinements, the evaluation of the refinements involves executing all of them; this can be replaced by a pack execution. In principle one could also perform several levels of refinement at this stage, adding all of them to A^*'s queue; part of the efficiency of A^* is then lost, but the pack execution mechanism is exploited to a larger extent. Which of these two effects is dominant will depend on the application: if most of the first-level refinements would be further refined anyway at some point during the search, clearly there will be a gain in executing a two-level pack; otherwise there may be a loss of efficiency. In any case, however, with single-level refinement packs a speedup should certainly be achieved.

4.2 Level-wise Frequent Pattern Discovery

An alternative family of data mining algorithms scans the refinement lattice in a breadth-first manner for queries whose frequency exceeds some user-defined threshold. The best-known instance of these level-wise algorithms is the APRIORI method for finding frequent item-sets [1]. WARMR [10] is an ILP variant of attribute-value based APRIORI.

Query packs in WARMR correspond to hash-trees of item-sets in APRIORI: both are used to store a subgraph of the total refinement lattice down to level n. The paths from the root down to level $n-1$ in that subgraph correspond to frequent patterns. The paths from root to the leaves at depth n correspond to candidates whose frequency has to be computed. Like hash-trees in APRIORI, query packs in WARMR exploit massive similarity between candidates to make their evaluation more efficient. Essentially the WARMR algorithm starts with an empty query pack and iterates between pack evaluation and pack extension (see Figure 4). The latter is achieved by adding all potentially frequent refinements[2] of all leaves in the pack, i.e., adding another level of the total refinement lattice.

[2] Refinements found to be specialisations of infrequent queries cannot be frequent themselves, and are pruned consequently.

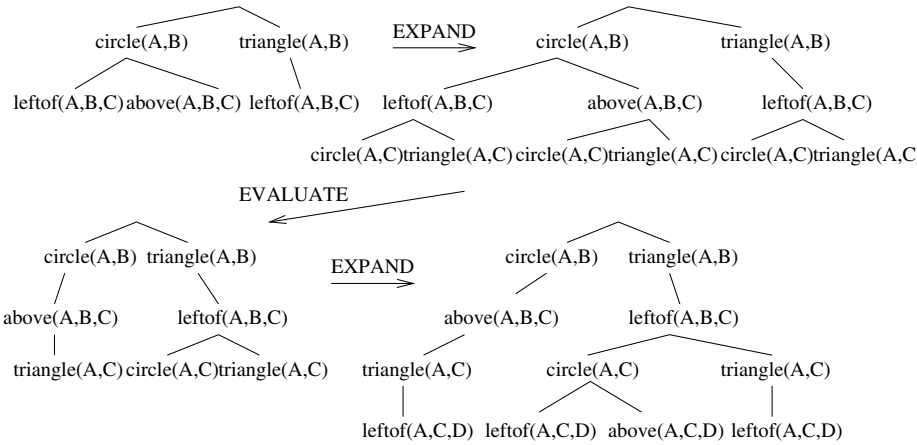


Fig. 4. A sequence of 4 query packs in WARMR. Refinement of the above left query pack results in the 3-level pack above right. Removal of queries found infrequent during pack evaluation results in the bottom left pack. Finally, another level is added in a second query expansion step to produce the bottom right pack. This iteration between expansion and evaluation continues until the pack is empty.

5 Experimental Evaluation

The goal of this experimental evaluation is to empirically investigate the actual speedups that can be obtained by re-implementing ILP systems so that they use the pack execution mechanism. At this moment such re-implementations exist for the TILDE and WARMR systems, hence we have used these for our experiments. We attempt to quantify both the speedup of packs w.r.t. to separate execution of queries, and the total speedup that this can yield for an ILP system.

5.1 Tilde

Comparison of Different Implementations of Tilde It is useful to consider four different ways in which TILDE can be run in its ILPROLOG implementation:

1. No packs: the normal implementation of TILDE, where queries are generated one by one and each is evaluated on all relevant examples (original implementation as described in [4]). Since queries are represented as terms, each evaluation of a query involves a meta-call in Prolog.
2. An "Examples in outer loop" implementation of TILDE, where examples are considered one by one and for each example all queries are run one after another (see [5]). Each call of a query, as in the previous setting, involves a meta-call.
3. Disjoint execution of packs: a query pack is executed in which all queries in the pack are put beside one another; i.e., common parts are not shared by

the queries. The computational redundancy in executing such a pack is the same as that in executing all queries one after another; the main difference is that in this case all queries are compiled.

4. Packed execution of packs: a query pack is executed where queries share as much as possible.

Of these four settings the second one turned out to be the slowest by far; we attribute this to the fact that queries are not compiled but executed using meta-calls. The most interesting information, however, is obtained by comparing the other three settings:

- The difference between packed execution and disjoint execution accurately indicates the efficiency gain that is obtained by redundancy removal (not blurred by any effects of pre-compilation and other possible implementation differences).
- The difference between the packed execution and the original implementation gives an idea of the net efficiency gain that is obtained, taking into account all implementation changes.

In the remainder of this section we focus on the timings for settings 1,3 and 4, which provide us with the above information.

Next to the total time needed to build a tree, we have also recorded the time needed to compile and load the query packs, and the time needed to execute them (this is excluding the time needed for computing statistics from the result set and retrieving the best query afterwards). To evaluate the net speedup of TILDE due to query pack execution, total times should be compared; when comparing disjoint and packed execution it is better to look at execution times only.

In a first experimental setup we measured times for TILDE on Bongard datasets[3] of different sizes, where no simple classification theory existed for the examples. In a second setup a true classification theory of small size existed for the examples, such that in most cases the same tree is learned; this mostly removes the influence of different tree sizes on induction times. In some cases there is a difference in tree size even for the same settings : different implementations may select different test when multiple tests of the same quality exist, which might give rise to significant differences in tree size (this was observed in earlier implementations of TILDE). We report these tree sizes in our tables. As can be seen in the table, in only one case a small difference in size was observed between trees induced under the same settings.

The size of a pack depends on how many refinements are generated at a certain node; this was varied by setting TILDE's lookahead parameter to 0, 1 or 2 (with 2 generating the largest packs).

Table 1 gives an overview of the experimental results we obtained for the first setup (trees of very different sizes), Table 2 gives a similar overview for the second

[3] The "Bongard" data sets contain problems related to those used by M. Bongard [6] for research on pattern recognition and were introduced as an ILP benchmark by [9].

LA	tree size (nodes)	original	disjoint total	disjoint comp	disjoint exec	packed total	packed comp	packed exec	speedup ratio *net*	speedup ratio **exec**
				592 examples						
0	27	*2.0*	*4.2*	2.26	**0.52**	*1.79*	0.35	**0.12**	*1.1*	**4.3**
1	15	*4.95*	*9.13*	5.06	**1.54**	*3.04*	0.77	**0.21**	*1.6*	**7.3**
2	11	*15.12*	*17.81*	9.12	**4.06**	*6.01*	1.79	**0.18**	*2.5*	**22.6**
				1194 examples						
0	58	*6.29*	*11.97*	6.44	**1.92**	*5.1*	0.98	**0.57**	*1.2*	**3.4**
1	54	*24.14*	*43.25*	24.79	**9.41**	*11.92*	3.12	**1.23**	*2.0*	**7.7**
2	22	*56.42*	*99.14*	65.76	**15.44**	*23.46*	8.45	**0.84**	*2.4*	**18.4**
				1804 examples						
0	48	*7.89*	*12.97*	6.02	**2.75**	*6.3*	1.13	**1.04**	*1.25*	**2.6**
1	58	*35.67*	*54.57*	27.76	**15.5**	*17.42*	4.46	**3.27**	*2.0*	**4.7**
2	13	*46.17*	*56.69*	27.34	**17.2**	*16.55*	4.87	**1.35**	*2.8*	**12.7**
				2986 examples						
0	65	*17.62*	*26.68*	11.39	**7.71**	*13.85*	1.92	**4.40**	*1.27*	**1.8**
1	80	*74.56*	*103.1*	41.49	**41.84**	*33.22*	5.49	**10.6**	*2.2*	**3.9**
2	83	*426.52*	*699.9*	406.6	**201.27**	*140.24*	45.95	**14.43**	*3.0*	**13.9**
				6013 examples						
0	129–131	*50.62*	*77.59*	22.52	**34.99**	*50.1*	3.93	**26.69**	*1.01*	**1.3**
1	148	*277.93*	*498.38*	122.15	**319.79**	*177.66*	14.07	**114.49**	*1.56*	**2.8**
2	81	*1600*	*2458*	539.64	**1752**	*386.8*	47.58	**187.66**	*4.1*	**9.3**

Table 1. Statistics of trees built by Tilde in function of the lookahead setting (LA = 0, 1, 2) and the "packs" setting (original = no packs, disjoint execution, packed execution). Reported times are the total time needed to build a tree, the time spent on compilation of packs and the time spent on their execution. Times are measured in seconds, tree sizes in nodes.

setup. The total induction time is reported, as well as (for pack-based execution mechanisms) the time needed for pack compilation and pack execution.

Some interesting observations are:

- Comparing total times, net speedup factors of 1 to 4.7 are obtained. The speedup becomes larger when larger packs are generated (higher lookahead setting).
- Comparing execution times of disjoint and packed execution, speedup factors of 1.3 up to 23 are obtained. Again, larger packs yield larger speedups.
- When subtracting the compilation time from the total induction time, it can be seen that disjoint execution is approximately as fast as the original implementation; i.e., the re-implementation from "queries in outer loop" to "examples in outer loop" does not have a net effect. Packed execution, of course, does yield an important gain.
- For each individual setting, the total induction time varies with the number of examples in the same way as for the other settings. This confirms that the re-implementation does not affect the way in which the complexity of the learning algorithm depends on the number of examples.

LA	tree size (nodes)	original	disjoint total	disjoint comp	disjoint exec	packed total	packed comp	packed exec	speedup ratio *net*	speedup ratio **exec**
			521 examples							
0	2	*0.63*	*1.05*	0.43	**0.22**	*0.58*	0.12	**0.06**	*1.09*	**3.7**
1	4	*1.7*	*1.26*	0.24	**0.51**	*0.74*	0.13	**0.07**	*2.30*	**7.3**
2	3	*4.6*	*2.94*	0.49	**1.42**	*1.26*	0.17	**0.12**	*3.65*	**11.8**
			1007 examples							
0	7	*1.12*	*1.11*	0.3	**0.32**	*0.73*	0.12	**0.12**	*1.53*	**2.7**
1	6	*4.06*	*3.08*	0.77	**1.31**	*1.59*	0.27	**0.27**	*2.55*	**4.9**
2	4	*12.74*	*7.38*	1.19	**3.86**	*3.05*	0.46	**0.34**	*4.18*	**11.4**
			1453 examples							
0	5	*1.44*	*1.28*	0.15	**0.49**	*0.95*	0.06	**0.21**	*1.52*	**2.3**
1	4	*5.25*	*3.48*	0.22	**2.05**	*1.85*	0.1	**0.56**	*2.84*	**3.7**
2	3	*14.33*	*8.83*	0.48	**5.66**	*3.49*	0.23	**0.68**	*4.11*	**8.3**
			2473 examples							
0	9	*3.34*	*2.99*	0.38	**1.33**	*2.21*	0.14	**0.75**	*1.51*	**1.8**
1	6	*12.3*	*9.11*	0.72	**6.08**	*4.78*	0.25	**2.14**	*2.57*	**2.8**
2	4	*40.05*	*24.66*	1.36	**17.94**	*8.44*	0.47	**2.64**	*4.75*	**6.8**
			4981 examples							
0	13	*8.46*	*7.92*	0.75	**4.18**	*6.2*	0.14	**3.1**	*1.36*	**1.4**
1	6	*35.63*	*29.45*	0.72	**23.76**	*16.63*	0.25	**11.4**	*2.14*	**2.1**
2	4	*116.93*	*84.14*	1.36	**71.41**	*25.43*	0.52	**13.64**	*4.60*	**5.2**

Table 2. Statistics of trees built by Tilde in function of the lookahead setting (LA = 0, 1, 2) and the "packs" setting (no packs, disjoint execution, packed execution). Reported times are the total time needed to build a tree, the time spent on compilation of packs and the time spent on their execution. Times are measured in seconds, tree sizes in nodes.

We have also run experiments on the Mutagenesis data set [15], with somewhat different results, as can be seen in Table 3. Query packs are much larger than for the Bongard data set; with a lookahead of 2 the largest packs had over 12000 queries. For these large packs a significant amount of time is spent compiling the pack, but even then large net speedups are obtained, in one specific case up to a factor of 20. Such high speedups can be obtained in cases where occasionally a computationally complex clause is generated (with many variables and many literals sharing variables) and then refined; a large proportion of the total induction time may be spent for this, and it is exactly here that the highest speedup factor can be expected (high c and n, see complexity analysis).

Comparison with Other Engines Since ILPROLOG is a new Prolog engine, we wanted to compare its performance with existing engines; obviously the efficiency gain achieved through the query pack execution it offers should not be offset by a less efficient implementation of the engine itself.

TILDE was run on the above data using the SICSTUS and MASTERPROLOG engines. The implementation of TILDE for these engines is almost exactly the

LA	tree size (nodes)	original	disjoint total	disjoint comp	disjoint exec	packed total	packed comp	packed exec	speedup ratio *net*	speedup ratio **exec**
Regression, 230 examples										
0	12	*5.45*	*7.84*	2.71	**3.04**	*4.35*	0.71	**1.68**	*1.25*	**1.81**
1	12	*22.71*	*36.33*	15.73	**16.6**	*12.18*	4.0	**4.85**	*1.86*	**3.42**
2	13	*239.84*	–	(99.87)	(**124.55**)	*72.06*	35.8	**19.86**	*3.33*	>**6.27**
Classification, 230 examples										
0	18	*6.03*	*9.04*	2.98	**4.05**	*4.51*	0.48	**2.32**	*1.34*	**1.75**
1	17	*28.38*	*42.4*	15.67	**21.67**	*13.6*	2.36	**7.38**	*2.09*	**2.94**
2	24	*2453.46*	–	(160.5)	(**397.41**)	*124.15*	39.29	**39.8**	*19.8*	>**9.99**

Table 3. Timings for Mutagenesis. A – in the table indicates that that run ended prematurely; timings between parentheses indicate times already consumed at that moment.

	Sicstus MasterProLog	ilProlog(original)	ilProlog(packs)
Bongard-592	52.3	10.05	14.87	6.13
Bongard-1194	206	37.5	56.98	24.78
Bongard-1804	217	27.62	46.57	15.06
Bongard-2986	1979	244	429	129
Bongard-6013	8600	651	1586	371
Bongard-521	16.7	3.14	4.49	1.37
Bongard-1007	58.6	8.23	12.7	3.5
Bongard-1453	105	8.91	14.2	3.37
Bongard-2473	228	20.45	39.2	9.33
Bongard-4981	714	43.45	115.4	27.7

Table 4. ilProlog compared to other engines (times in seconds)

same as the one for the ilProlog engine when the "original" setting (no packs) is used. Table 4 shows some results. It can be seen from the table that at this moment, and on these data, ilProlog is less efficient than MasterProLog, however the pack execution mechanism amply compensates for that.

Some remarks are to be made here: a) the Leuven ILP systems were originally implemented in MasterProLog and these implementations make use of some non-standard builtins which for the other Prolog systems have to be simulated, which puts MasterProLog at an advantage when comparing timings; b) ilProlog is still under construction and further efficiency improvements in the engine are expected (for instance, its compiler does not produce native code yet, which the other compilers do); c) our experience with these experiments suggests that timings are relatively unstable w.r.t. certain implementation changes, especially with respect to dynamically asserted information; therefore these results (in particular for Sicstus) have to be taken with a grain of salt.

Mutagenesis

Level	Queries	Frequent queries
1	1	1
2	9	6
3	27	20
4	90	72
5	970	897

Table 5. Number of queries for the Mutagenesis experiment with WARMR.

Level	With packs		No packs ILPROLOG		No packs MASTERPROLOG		speedup ratio	
	exec	total	exec	total	exec	total	*net*	**exec**
2	0.65	0.93	1.17	1.26	0.37	0.62	*1.35*	**1.80**
3	4.63	6.69	8.24	9.2	1.62	2.77	*1.38*	**1.78**
4	40.14	65.22	94.41	107.04	10.46	24.25	*1.64*	**2.35**
5	345.75	673.75	861.77	1078.09	101.52	311	*1.60*	**2.49**

Table 6. Results on Mutagenesis

5.2 Warmr

Used Implementations and Engines For WARMR we consider the following implementations:

1. No packs: the normal implementation of WARMR, with "queries in outer loop", where queries are generated and evaluated one by one.
2. With packs: An "examples in outer loop" implementation where first all queries for one level are generated and put into a pack, and then this pack is evaluated on each example.

For the "no packs" implementation we also give the execution times for MASTERPROLOG.

Datasets We used the Mutagenesis dataset, with a language bias that was chosen so as to generate a limited number of refinements (i.e., a relatively small branching factor in the search lattice); this allows us to generate query packs that are relatively deep but narrow. Table 5 summarises the number of queries for each level.

Results In Table 6 the execution times of WARMR on Mutagenesis are given, with maximal search depth varying from 2 to 5 levels.

These results raised our interest because they show a behaviour that is quite different from the one observed with TILDE. Speedup factors barely increase with increasing depth of the packs, in contrast to TILDE where larger packs yielded higher speedups. At first sight we found this surprising; however it becomes less so when the following observation is made. When refining a pack into a new pack

by adding a level, WARMR prunes away branches that lead only to infrequent queries. There are thus two effects when adding a level to a pack: one is the widening of the pack at the lowest level (at least on the first few levels, a new pack typically has more leaves than the previous one), the second is the narrowing of the pack as a whole (because of pruning). Since the speedup obtained by using packs largely depends on the branching factor of the pack, speedup factors can be expected to decrease when the narrowing effect is stronger than the widening-at-the-bottom effect. We suspect both effects are approximately equally strong here. We had not anticipated this behaviour and think it deserves further investigation.

5.3 Summary of Experimental Results

Our experiments confirm that a) query pack execution in itself is much more efficient than executing many highly similar queries separately; b) existing ILP systems (we use TILDE and WARMR as examples) can use this mechanism to their advantage, achieving significant speedups (up to a factor 20 in our experiments); and c) although a new Prolog engine is needed to achieve this, the current state of development of this engine is already such that more advanced engines that do not have the query pack mechanism cannot compete with it.

In addition, differences between the effect of pack execution on TILDE and WARMR have come to light, prompting further investigation of, e.g., the influence of size and shape of packs on the efficiency gains obtained with them.

6 Related Work

The re-implementation of TILDE is related to the work by [12] who were the first to describe the "examples in outer loop" strategy for decision tree induction. The query pack execution mechanism, here described from the Prolog execution point of view, can be seen as a first-order counterpart of APRIORI's mechanism for counting item-sets [1].

The idea of optimising sets of queries instead of individual queries has existed for a while in the database community; Tsur et al. [16] describe an algorithm for efficient execution of so-called query *flocks* in this context. Like our query pack execution mechanism, the query flock execution mechanism is inspired by APRIORI and is set in a deductive database context. The main difference between our query packs and the query flocks described in [16] is that query packs are more hierarchically structured and the queries in a pack are much less similar than the queries in a flock. (A flock is represented by a single query with placeholders for constants, and is equal to the set of all queries that can be obtained by instantiating the placeholders to constants. Flocks could not be used for the applications we consider here.)

7 Conclusions

There is a lot of redundancy in the computations performed by most ILP systems. In this paper we have identified a source of redundancy and proposed a

method for avoiding it: execution of query packs, and we have discussed how query pack execution can be incorporated in ILP systems. The query pack execution mechanism itself has been implemented in a new Prolog system called ILPROLOG dedicated to data mining tasks, and two ILP systems have been re-implemented to make use of the mechanism. We have experimentally evaluated these re-implementations, and the results of these experiments confirm that large speedups may be obtained in this way. We conjecture that the query pack execution mechanism can be incorporated in other ILP systems and that similar speedups can be expected.

The problem setting in which query pack execution was introduced is very general, and allows the technique to be used for any kind of task where many queries are to be executed on the same data, as long as the queries can be organised in a hierarchy.

Future work includes further improvements to the ILPROLOG engine and the implementation of techniques that will increase the suitability of the engine to handle large data sets. Another interesting issue is how to port the proposed query pack execution mechanism to the context of relational databases. A lot of work has been done in the database community concerning optimisation of single queries, but optimisation of sets of queries is relatively new. In the best case one might hope to combine the efficient bottom-up computations usually performed in databases with a computation sharing mechanism such as the one we here propose. Such a result would significantly increase the efficiency with which large databases can be mined.

Acknowledgements

Hendrik Blockeel is a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders. Luc Dehaspe is a post-doctoral fellow of the K.U.Leuven Research Fund. Jan Ramon is funded by the Flemish Institute for the Promotion of Scientific Research in Industry (IWT). Henk Vandecasteele is supported by the FWO-project G.0246.99 "Query languages for database mining". The authors would like to thank Luc De Raedt for his influence on this work, and Ashwin Srinivasan for suggesting the term "query packs".

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. The MIT Press, 1996.

[2] H. Blockeel. *Top-down induction of first order logical decision trees.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1998. `http://www.cs.kuleuven.ac.be/~ml/PS/blockeel98:phd.ps.gz`.

[3] H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.

[4] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
[5] H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.
[6] M. Bongard. *Pattern Recognition*. Spartan Books, 1970.
[7] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996. See also `http://www.cs.sunysb.edu/~sbprolog`.
[8] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
[9] L. De Raedt and W. Van Laer. Inductive constraint logic. In Klaus P. Jantke, Takeshi Shinohara, and Thomas Zeugmann, editors, *Proceedings of the 6th International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
[10] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1):7–36, 1999.
[11] Bart Demoen, Gerda Janssens, and Henk Vandecasteele. Executing query flocks for ILP. In Sandro Etalle, editor, *Proceedings of the Eleventh Benelux Workshop on Logic Programming*, Maastricht, The Netherlands, November 1999. 14 pages.
[12] M. Mehta, R. Agrawal, and J. Rissanen. SLIQ: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, 1996.
[13] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13, 1995.
[14] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann, 1993.
[15] A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, 1995.
[16] Dick Tsur, Jeffrey D. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov, and Arnon Rosenthal. Query flocks: A generalization of association-rule mining. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 1–12, New York, June 1–4 1998. ACM Press.

A Logical Database Mining Query Language*

Luc De Raedt

Institut für Informatik, Albert-Ludwig-University
Universitätsgelände Flugplatz
D-79085 Freiburg, Germany
deraedt@informatik.uni-freiburg.de

Abstract. A novel query language for database mining, called RDM, is presented. RDM provides primitives for discovering frequent patterns and association rules, for manipulating example sets, for performing predictive and descriptive induction and for reasoning about the generality of hypotheses. RDM is designed for querying deductive databases and employs principles from inductive logic programming. Therefore RDM allows to query patterns that involve multiple relations as well as background knowledge. The embedding of RDM within a programming language such as PROLOG puts database mining on similar grounds as constraint programming. An operational semantics for RDM is outlined and an efficient algorithm for solving RDM queries is presented. This solver integrates Mitchell's versionspace approach with the well-known APRIORI algorithm by Agrawal *et al.*
Keywords : database mining query language, inductive logic programming, relational learning, inductive database, APRIORI, versionspaces.

1 Introduction

Imielinski and Mannila [11] present a database perspective on knowledge discovery. From this perspective, knowledge discovery is regarded as a querying process to a database mining system. Querying for knowledge discovery requires an extended query language (w.r.t. database languages), which supports primitives for the manipulation, mining and discovery of rules, as well as data. The integration of such rule querying facilities provides new challenges for database technology.

Mannila and Toivonen [14] formulate the general pattern discovery task as follows. Given a database r, a language $\mathcal{L}$ for expressing patterns, and a selection predicate q, find the theory of r with respect to $\mathcal{L}$ and q, i.e. $Th(\mathcal{L}, r, q) = \{\phi \in \mathcal{L} \mid q(r, \phi)\ is\ true\}$. This formulation of pattern discovery is generic in that it makes abstraction of several specific tasks including the discovery of association rules, frequent patterns, inclusion dependencies, functional dependencies,

* An early extended abstract of this paper appeared as part of [7]. The present paper expands significantly on this work. The work was also presented at the JICSLP Workshop in Manchester 1998, and the German Machine Learning Workshop, Magdeburg, 1999.

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 78–92, 2000.

frequent episodes, ... Also, efficient algorithms for solving these tasks are known (cf. [14]).

So far, the type of selection predicate that has been considered is simple and typically relies on the frequency of patterns. However, knowledge discovery is often regarded as a cyclic and iterative process (cf.[10]), where one will consider a number of different selection predicates, patterns, example sets and versions of the database. Therefore, there is a need for considering and combining different theories $Th(\mathcal{L}_i, q_i, r_i)$. This practical necessity is also the main motivation underlying the work on inductive databases [15] and database mining query languages [11,17]. Indeed, a number of extensions to database languages such as SQL have been introduced with the aim of supporting the generation and manipulation of the theories $Th(\mathcal{L}_i, q_i, r_i)$.

This paper presents a novel query language, called RDM − Relational Database Mining, as well as its execution mechanism. RDM supports a larger variety of selection predicates than previous approaches. In addition to providing an operator to manipulate patterns, it also allows to manipulate sets of examples, and provides selection predicates that impose a minimum or maximum frequency treshold. This allows RDM to address descriptive as well as predictive induction. In descriptive induction one is interested in frequent patterns, whereas in predictive induction, one aims at discovering patterns that are frequent on the positive examples but infrequent on the negative ones. Further primitives in RDM are based on a notion of generality (among patterns) and on coverage (w.r.t. specific examples). Queries in RDM will consist of a number of primitives. Each primitive will impose a specific constraint on the patterns to be discovered and will result in a theory $Th(\mathcal{L}_i, q_i, r_i)$. RDM will then efficiently compute the intersection of all these theories. RDM's execution mechanism integrates the versionspace approach by Tom Mitchell [19] with the levelwise algorithm in APRIORI [1]. Directly computing the intersection of the theories contrasts with the approach taken in the MINE RULE operator by Meo *et al.* [17] and the approach of Boulicaut *et al.* [2] in that these latter approaches typically generate one theory $Th(\mathcal{L}_i, q_i, r_i)$ and then repeatedly modify it.

As RDM aims at providing a general query language for database mining, it employs patterns in the form of DATALOG queries [5,6,4] as in the field of relational learning or inductive logic programming [20]. This allows RDM to mine multiple relations in a database at once and to express patterns beyond the scope of propositional approaches (as shown by [6]). Embedding RDM within a programming language such as PROLOG [3] puts database mining on the same methodological grounds as constraint programming. Indeed, each of the different selection predicates or primitives in a query or program imposes certain constraints on the patterns. As in constraint programming, we have to specify the semantics of these primitives as well as develop efficient solvers for queries and programs. An operational semantics for RDM will be specified and an efficient solver will be presented.

The paper is organised as follows : in Section 2, we define the basic RDM primitives based on logic, in Section 3, we show RDM at work through a number

of database mining queries, in Section 4, we present a solver for simple queries, in Section 5, we discuss how to extend the language and solver, and finally, in Section 6, we conclude and touch upon related work.

2 Terminology

2.1 DATALOG

As [5,4] we will use DATALOG (e.g. [6]) to represent both data and patterns. In DATALOG, *terms* are either constants (written in lowercase) or variables (starting with a capital). An *atom* is then of the form $p(t_1, ..., t_n)$ where p is a predicate (or relation) symbol of arity n and the t_i are different terms. A *definite clause* is a universally quantified formula of the form $H : -A_1, ..., A_n$ where H and the A_i are different atoms. The formula can be read as H is A_1 and ... and A_n. If $n = 0$ then the clause is a *fact* (this corresponds to a tuple in a relational database). A (deductive) database is then a set of definite clauses. A subsitution $\theta = \{V_1 = t_1, ..., V_k = t_k\}$ is an assignment of terms t_i to variables V_i. A substitution θ can be applied to a clause or query c resulting in $c\theta$ where all variables V_i have been substituted by the corresponding terms t_i. DATALOG queries are of the form : $-B_1, ..., B_m$ where the B_i are atoms. Using resolution theorem provers, one can answer a query q by generating all answersubstitutions θ for which $q\theta$ follows from the database. It will be convenient to represent the set of all answersubstitutions for a query q and a database r as $answerset(q, r)$. Sometimes we will only be interested in the projection $\theta_W = \{V_i = t_i \mid V_i \in W\}$ of a substition $\theta = \{V_1 = t_1, ..., V_k = t_k\}$ on a set of variables $W \subset \{V_1, ..., V_k\}$ or in the projection $answerset_W(q, r) = \{\theta_W \mid \theta \in answerset(q, r)\}$ of an $answerset(q, r)$.

2.2 RDM Primitives

Definition 1. *A* pattern $\phi(Q, K)$ *consists of a DATALOG query* Q *and a set of variables* K*, called the key, occurring in* Q*.*

The idea is that the key represents the identifier of the entities being considered. The key thus corresponds to the notion of example and will reflect what is being counted. This is similar to the GROUP BY phrase in the MINE RULE operator of [17]. One pattern about transactions involving beer and sausage is e.g. ϕ(`(beer(X),sausage(X)), {X})`. Notice that we ommit the ':-' of the DATALOG query for readability reasons.

Data mining with RDM is then the search for patterns that satisfy certain requirements. Three primitives are foreseen to specify requirements in RDM. The first concerns the frequency of patterns, which is defined as follows (following [5,4]):

Definition 2. $frequency(\phi(Q, K), r) =\mid answerset_K(Q, r) \mid$ *where* $\phi(Q, K)$ *is a pattern and* r *a database.*

The frequency is defined as the number of substitutions for the key for which the query succeeds. This primitive allows to impose constraints on the frequency of the patterns, e.g. $frequency(\phi(Q,K),r) > min$ or $frequency(\phi(Q,K),r) < max$, where Q and K are variables (as we shall illustrate below).

The second primitive concerns the syntactic form of queries in patterns. The primitive $\preceq$ specifies the relation of the pattern to specified queries.This corresponds to encoding the *is more general than* relation.

Definition 3. *$Q_1 \preceq Q_2$ if and only Q_1 is more general than Q_2.*

We will also be using $\prec$ which has the classical semantics. In the case of using DATALOG patterns, $\preceq$ can best be implemented by Plotkin's θ-subsumption [21], which states that Q_1 is more general than Q_2 if and only if there exists a substition θ such that $Q_1\theta \subseteq Q_2$. E.g. the query `(beer(X), diapers(X))` θ-subsumes the pattern `(beer(Y), diapers(Y), sausage(Y))` with $\theta = \{X = Y\}$. This primitive allows to express which patterns one is interested in.

The third primitive concerns the notion of an example. An example in RDM corresponds to a substitution θ that substitutes all variables in the key K with constants in the pattern $\phi(Q,K)$. Using this primitive, it is possible to specify that a pattern should cover a specific example :

Definition 4. *An example θ is covered by a pattern $\phi(Q,K)$ and a database r, notation $r \models Q\theta$, if and only if the query $Q\theta$ succeeds in the database r.*

We will also be using the notation $r \not\models Q\theta$ to reflect the fact that the example θ is not covered. E.g. given a database r containing the facts

```
diapers(t133), beer(t133), sausage(t133)
```

we have that the pattern ϕ(`(beer(T), sausage(T))`, {`T`}) covers the example {`T = t133`} because the query `:- beer(t133), sausage(t133)` succeeds in the database.

Finally, RDM provides a number of extensions of the above primitives in order to provide more flexibility. The first extension allows RDM to work with sets of examples E and to extend the frequency and coverage primitives to work with sets of examples. This results in the following definitions :

Definition 5. *$frequency(\phi(Q,K),r,E) = \mid answerset_K(Q,r) \cap E \mid$ where $\phi(Q,K)$ is a pattern, r a database, and E a set of examples.*

Using this extension, we can specify RDM queries to discover patterns that are frequent on a set of positive examples but infrequent on a set of negative examples.

Definition 6. *A set of examples E is covered by a pattern $\phi(Q,K)$ and a database r, notation $r \models E$, if and only if $frequency(\phi(Q,K),r,E) = \mid E \mid$.*

Similarly, we will say that a set of examples E is not covered if the frequency is 0.

The second extension allows to employ negation in various forms, e.g. to specify that a query is not more specific than a given other query e.g. `not Q` $\preceq$ `(sausage(X), beer(X))`.

The final extension concerns embedding the primitives in a high-level declarative programming language. Doing this puts data mining on the same methodological grounds as constraint programming. For the purposes of RDM, it seems most appropriate to embed its primitives within the PROLOG programming language because RDM assumes DATALOG as the underlying database model.

Further advantages of embedding RDM within PROLOG are that this naturally allows us to formulate complex RDM queries. E.g. the query

`?- Q` $\preceq$ `(beer(X), diapers(X), mustard(X), sausage(X)),`
$frequency(\phi(Q, \{X\}), r) >$ `10 .`

would return all patterns involving beer, diapers, mustard and sausage, whose frequence on database r exceeds 10 where the key is given by the {X}. This shows that the embedding of RDM within PROLOG allows to use variables (written in capitals) that range over the query-part of a pattern (such as Q). Integrating PROLOG's and RDM's execution mechanism allows us to get answers for these variables. One such answer might be `Q = (beer(X), diapers(X), mustard(X))` stating that for this particular instantiation of the variable Q all the primitives hold.

The embedding also allows us to define complex predicates in terms of the primitives. E.g. asssociation rules can now be defined as follows in PROLOG.

`associationrule(` ϕ`(Condition,K),` ϕ`(Conclusion,K), R, Acc, Freq) :-`
`Condition` $\prec$ `Conclusion,`
`Freq is` $frequency$`(`ϕ`(Conclusion,K),R),`
`Acc is` $frequency$`(`ϕ`(Condition,Key),R) / Freq.`

The predicate `associationrule` will produce association rules of the database `R` of the form `Conclusion if Condition` and will compute their accuracy and frequency. E.g. the query

`?- C` $\preceq$ `(beer(X), diapers(X), mustard(X), sausage(X)) ,`
`associationrule(` ϕ`(Q,{X}),` ϕ`(C,{X}), r, Acc, Freq) ,`
`Freq > 10, Acc > 0.5 .`

would return all association rules with a frequency on `r` of more than 10 and an accuracy of more than 50 per cent. E.g. the pattern `beer(X),diapers(X) if beer(X)`, which for this case could be simplified by ommitting `beer(X)` in the conclusion part.

This type of query is similar to the queries one can express within Imielinski and Virmani 's M-SQL [12]. Using this same type of queries it is possible to

emulate the essence of Dehaspe and De Raedt's WARMR [4] and De Raedt and Dehaspe's Claudien [8].

Observe also that it is easy to enhance RDM by providing further primitives such as e.g. $sample(E, P)$ with E an example set and P a probability, which would generate a sample of the examples (drawn according to probability P). This and other extensions are however quite straightforward, and merely provide further syntactic sugar. Therefore we will not discuss such extensions any further here.

So far, we have defined a semantics for RDM and embedded it within PROLOG. It is straightforward to implement the *specification* of the primitives within PROLOG (cf. [7] for such a high level implementation). However, a direct implementation will not work efficiently, and therefore a more advanced implementation will be sketched in Section 4. It should also be mentioned that not all database mining queries are safe, but that it is possible to specify safe queries using modedeclaration (as typically done in Prolog manuals). For safe execution, it is e.g. required that for all patterns ϕ`(Q,K)` searched for the data mining query in RDM - PROLOG must include $Q \preceq (lit_1, ..., lit_n)$ before any further reference to Q is made. This is necessary in order to specify the range of DATALOG queries searched for. Also, without bounding Q in this manner, many of the predicates defined above would not operate correctly. We will not go into further (implementation) details of the semantics at this point.

3 Data Mining with RDM

We now basically have all primitives needed to generate database mining queries. In this section, we demonstrate the expressive power of RDM by examples.

First, as already sketched above, one can naturally discover all frequent patterns bounded from below by some DATALOG query. The RDM query shown earlier demonstrated this for the case of item-sets. Due to use of first order language for expressing patterns, it is also possible to simulate the behaviour of the WARMR system [4,5]. E.g. the query

```
?- (atomel(C,A1,c), atomel(C,A2,h), bond(C,A1,A2)) ⪯ Q,
Q ⪯ (atomel(C,A1,c), atomel(C,A2,h), bond(C,A1,A2),
atomtype(A1,t1), ..., atomtype(A1,tn), atomtype(A2,t1),
atomtype(A2,t2)), frequency(φ(Q,{C}),r) > 10 .
```

could be used to find frequent patterns in molecules that involve bonds in which a carbon and a hydrogen atom occur and further properties of the atoms (such as types) are specified.

Further details of the use of such first order patterns and the relation to existing frameworks for frequent pattern discovery are provided in a recent paper by Dehaspe and Toivonen [6]. They show that nearly any type of frequent pattern considerd in the data mining literature can be expressed in the presented first order framework. Hence, it is easy to simulate these in RDM.

A second type of query is to generate association rules as sketched in section 2.

The third type of queries is aimed at predictive data mining instead of descriptive datamining. It therefore employs both positive and negative examples. E.g.

```
?- Q ⪯ (Item-1(X), ... , Item-n(X)),
N is frequency(φ(Q,{X}),r,NegSet ),
N< 10,
P is frequency(φ(Q,{X}),r,PosSet),
Acc is P / (P+N), Acc > 0.9.
```

This query would generate condition parts of rules that are at least 90 per cent correct, and that cover at most 10 negative examples in the $NegSet$ (which we did not represent explicitly here). Once such rules are found, they can (partly) define a predicate with the key as argument(s). E.g. all DATALOG queries Q generated may be turned into clauses `positive(X) :- Q` that could be asserted in the database for further use. This is akin to what many classification systems search for. Also, rather than involving a minimum frequency, this type of rule querying imposes a maximum frequency (on the negative examples).

Fourthly, sometimes predictive data mining approaches employ seed examples. These seed examples are used to constrain the rules searched for. E.g. when positive examples are used as seeds, the induced rule should cover the positive. This can easily be expressed with the covers primitive. In typical database mining situations, these different constructs will be combined in order to find relevant (sets of) patterns. E.g.

```
?- Q ⪯ (item-1(T),.., item-n(T)),
r ⊨ Q{T = t190}, positive seed
r ⊭ Q {T = t133}, negative seed
P is frequency(φ(Q,{T}),r,PosSet),
N is frequency(φ(Q,{T}),r, NegSet),
N< 10, Acc is P / (P + N), Acc > 0.9 .
```

So far, we have only allowed queries that involve a single pattern. However, it may be useful to search for multiple patterns simultaneously. E.g.

```
?- Q ⪯ (item-1(T),.., item-n(T)),
P is frequency(φ(Q,{T}),r,PosSet),
N is frequency(φ(Q,{T}),r, NegSet),
Acc is P / (P + N), Acc > 0.9,
coveredexamples(φ(Q,{T}), r, PosSet, Covered),
delete(Covered,PosSet,Uncovered),
Q2⪯ (item-1(T),.., item-n(T)),
P2 is frequency(φ(Q2,{T}),r,Uncovered),
P2 > 100.
```

This query would first find a DATALOG query Q that is 90 per cent correct on the example set and then try to find a pattern Q2 whose frequency exceeds 100 on the positive examples not covered by Q.

Of course, the most expressive way of using the query language is to embed the primitives in simple Prolog programs. Using this facility it is easy to emulate inductive logic programming systems (such as e.g. Quinlan's FOIL [22]) in a one page meta-program.

4 Answering Database Mining Queries

Obviously, plainly using the primitives as they have been defined will be desastrous from the efficiency point of view. This is because all patterns in the language would simply be enumerated using and tested afterwards. The question then is : how can we efficiently produce answers to the queries formulated above ?

Let us here concentrate on simple database mining queries. Simple queries search for a single pattern only.

The key insight that leads to efficient algorithms for answering database mining queries is that 1) the space of patterns is partially ordered by the $\preceq$ relation, and 2) that all primitives provided in RDM behave monotonically with regard to $\preceq$. Therefore each constraint on a pattern allows to prune the search. Furthermore, a set of constraints defines a versionspace of acceptable patterns. This versionspace is completely determined by its S and G sets (as defined by Mitchell 1982).

We have the following kind of basic constraints or information elements (the extensions can be dealt with analoguously) :

- $Query \preceq q$ (resp. $q \preceq Query$) which states that the target Query is more general (resp. more specific than a given query q; similar elements can be defined wrt. $\prec$.
- $frequency(\phi(Q, K), r) > t$ (resp. $\geq$) : which specifies a constraint on the frequency of the pattern (on the specified Data)
- $r \models Q\theta$ which expresses that the query should cover the example θ
- also, each of the above primitives can be negated

Information elements of the first type (i.e. concerning $\preceq$) are directly handled by Mellish's description identification algorithm [18], which extends Mitchell's versionspace algorithm. Fact is that the statements of the form $Query \preceq q$ (resp. $notQuery \preceq q$) can be interpreted as $Query$ covers the 'positive example' q (resp. 'negative example'). In this view the query q is regarded an example (in a different example space than that of RDM). Mellish's description identification algorithm extends Mitchell's well-known versionspace algorithm by handling statements of the form $q \preceq Query$ and its negation. These additional information elements are dual in the original ones, and so is its algorithmic treatment.

Thus it is straigthforward to process the elements concerning $\preceq$ with the versionspace algorithm. First define :

$glb(a,b) = \max\{d \in \mathcal{L} \mid a \preceq d \text{ and } b \preceq d\}$
$lub(a,b) = \min\{d \in \mathcal{L} \mid d \preceq a \text{ and } d \preceq b\}$
$mgs(a,b) = \max\{d \in \mathcal{L} \mid a \preceq d \text{ and } not(d \preceq b)\}$
$msg(a,b) = \min\{d \in \mathcal{L} \mid d \preceq a \text{ and } not(b \preceq d)\}$

The *max* and *min* functions will return the set of maximal (resp. minimal) elements within the partial order defined by $\preceq$. We can then use the following algorithm :

$S := \{\top\}$; $G := \{\bot\}$;
for all constraints i **do**
  case i of $q \preceq Query$:
    $S := \{s \in S \mid q \preceq s\}$
    $G := \max\{glb(q,g) \mid g \in G \text{ and } \exists s \in S : glb(q,g) \preceq s\}$
  case i of $Query \preceq q$:
    $G := \{g \in G \mid g \preceq q\}$
    $S := \min\{lub(q,s) \mid s \in S \text{ and } \exists g \in G : g \preceq lub(q,s)\}$
  case i of $not\ Query \preceq q$:
    $S := \{s \in S \mid not(s \preceq q)\}$
    $G := \max\{m \mid \exists g \in G : m \in mgs(g,q) \text{ and } \exists s \in S : m \preceq s\}$
  case i of $not\ q \preceq Query$:
    $G := \{g \in G \mid not(q \preceq g)\}$
    $S := \min\{m \mid \exists s \in S : m \in msg(s,q) \text{ and } \exists g \in G : g \preceq m\}$
  case i of $r \models Query\theta$
    $G := \{g \in G \mid r \models g\theta\}$
    $S := \min\{s' \mid r \models s'\theta \text{ and } \exists s \in S : s' \preceq s \text{ and } \exists g \in G : g \preceq s\}$
  case i of $r \not\models Query\theta$:
    $S := \{s \in S \mid r \not\models s\theta\}$
    $G := \max\{g' \mid r \not\models g'\theta \text{ and } \exists g \in G : g \preceq g' \text{ and } \exists s \in S : g' \preceq s\}$
  case i of $frequency(\phi(Query, Key), r) > f$:
    $G := \{g \in G \mid frequency(\phi(g, , Key), r) > f\}$
    $S := \min\{s' \mid frequency(\phi(s', Key), r) > f$ *and*
      $\exists s \in S : s' \preceq s \text{ and } \exists g \in G : g \preceq s'\}$
  case i of $frequency(\phi(Query, Key), r) < f$:
    $S := \{s \in S \mid frequency(\phi(s, Key), r) < f\}$
    $G := \max\{g' \mid frequency(\phi(g', Key,), r) < f$ *and*
      $\exists g \in G : g \preceq g' \text{ and } \exists s \in S : g' \preceq s\}$

When analysing the above algorithm, the glb, lub, mgs and msg operators can be straightforwardly and efficiently implemented. However, the key implementation issue is how to compute the S set for the case of $frequency(\phi(Q,K),r) > f$. Similar questions arise for the dual constraint and the use of covers. In the data mining literature, a lot of attention has been devoted to this, resulting in algorithms such as APRIORI that will only evaluate those patterns on the database that could possible be frequent. It turns out that we can find the S set for this case in two different manners, which correspond to traversing the space in two

dual ways. When working general to specific, one can integrate the basic ideas of the APRIOR approach into versionspaces.

We now sketch how this could be realized by looking at the case that a minimum frequency treshold is imposed on the patterns. The other cases however can be handled in a similar way.

Before specifying the two dual versions, we need a notion of a refinement and a generalization operator.

Definition 7. *A refinement operator* $\rho_s(Q) = max\{Q' \in \mathcal{L} \mid Q \prec Q'\}$, *where* Q *is a query. A generalization operator* $\rho_g(Q) = min\{Q' \in \mathcal{L} \mid Q' \prec Q\}$.

Let us then first sketch the top-down algorithm to update S and G when imposing a minimal frequency treshold.

$NewG := \{g \in G \mid frequency(g, r) > f\}$
$I := G - NewG$; *the set of infrequent patterns*
$F := NewG$; *the set of frequent patterns*
$L_0 := NewG$; $i := 0$; L_i *contains frequent queries for further refinement*
while $L_i \neq \emptyset$ **do**
  $L_{i+1} := \{p \mid \exists q \in L_i : p \in \rho_s(q)$ *and* $\exists s \in S : p \preceq s\}$
  $NewIs := \{p \mid p \in L_{i+1}$ *and* $\rho_g(p) \cap I \neq \emptyset\}$
  $I := NewIs \cup I$; $L_{i+1} := L_{i+1} - NewIs$;
  for all $q \in L_{i+1}$ **do**
    if $frequency(q, r) > f$
    then add q to F **else** add q to I and delete q from L_{i+1}
  $i := i + 1$;
endwhile
$S := min(F)$[1] ; $G := NewG$;

Let us first consider the case where $S = \{\bot\}$ and $G = \{\top\}$ and where we work with itemsets. In this case the refinement operator will merely add a single item to a query and the generalization operator will delete a single item from the itemset (in all possible manners). In this case, the above algorithm will behave roughly as the APRIORI algorithm. The L_i will then contain only itemsets of size i and the algorithm will keep track of the set of frequent item sets F as well as the infrequent ones [2]. The algorithm will then repeatedly compute a set of candidate refinements L_{i+1}, delete those item sets that cannot be frequent by looking at the frequency of its generalizations, and evaluate the resulting possibly frequent itemsets on the database. This process then continues until L_i becomes empty.

The basic modifications to run it in our context are concerned with the fact that we need not consider any element that is not in the already computed versionspace (i.e. any element not between an element of the G and the S set). Secondly, we have to compute the updated S set, which should contain all frequent elements whose refinements are all infrequent.

[2] Explicitly keeping track of I is not done by APRIORI, but is similar to some of its extensions, cf. [5]

Finding the updated G and S sets can also be realized in the dual manner. In this case, one will initialize L_0 with the elements of S that are infrequent and proceed otherwise completley dual. The resulting algorithm is shown below :

$G := \{g \in G \mid frequency(g, r) > f\}$
$FS := \{s \in S \mid frequency(s, r) > f\}$
$I := S - FS$ *the set of infrequent patterns*
$F := FS$;[3] *the set of frequent patterns*
$L_0 := I$; $i := 0$; L_i *contains infrequent queries for further generalization*
while $L_i \neq \emptyset$ **do**
    $L_{i+1} := \{p \mid \exists q \in L_i : p \in \rho_g(q)$ *and* $\exists g \in NewG : g \preceq p\}$
    $NewFs := \{p \mid p \in L_{i+1}$ *and* $\rho_s(p) \cap F \neq \emptyset\}$
    $F := NewFs \cup F$; $L_{i+1} := L_{i+1} - NewFs$;
    for all $q \in L_{i+1}$ **do**
        if $frequency(q, r) > f$
        then add q to F and remove from L_{i+1} **else** add q to I
    $i := i + 1$
endwhile
$S := min(F)$;

Simple queries query for a single pattern and the above algorithm shows how such queries can be answered. However, more complex queries would involve multiple patterns. The algorithm for computing answers to simple queries can be upgraded towards this type of queries. The easiest way to handle such queries is by processing the literals in the query from left to right (as in Prolog) and working with multiple versionspaces. Algorithms for realizing this more efficiently are however the subject of future work.

5 Extensions to the Basic Engine and Language

5.1 Optimization Primitives

Two primitives that seem especially useful are *min* and *max*. Indeed, one could imagine being interested in those patterns that satisfy a number of constraints and in addition have maximum frequency on a certain dataset or are minimally general. Let us therefore define :

Definition 8. *An optimization literal* $max(p(\phi(Q, K)), rel)$ *(resp. min) takes as argument a predicate* p *and a relation* rel *to be optimized. The predicate* p *imposes constraints on the pattern* $\phi(Q, K)$ *and* rel *specifies the criterion that should be maximized (resp. minimized). The criterion is either generality or* $frequency(\phi(Q, K), r, E)$*. The optimization literal then succeeds for those patterns satisfying the constraints imposed by* p *and being optimal w.r.t.* rel*.*

As an example consider the following query :

```
patt(φ(Q,K)) :-
      Q ⪯ (beer(X), diapers(X), mustard(X), sausage(X)),
      frequency(φ(Q,K),r) > 100.

?-min(pat(φ(Q,K)),frequency(φ(Q,K),r)).
```

This query would generate the pattern with minimum frequency (but higher than 100) over the items beer, diapers, mustard and sausage. For this particular query, one might want also to use two different databases (one within p and one within min).

Within the skecthed solver for simple queries, it is relatively easy to accomodate this type of constructs. Indeed, because the primitives w.r.t. frequency and generality behave monotonically w.r.t. $\preceq$, one only needs to consider the maximum (resp. minimum) elements in the solutions to p. So, to answer a query containing an optimization literal, one first computes the versionspace (i.e. the S and G sets) for p. If the criterion to be optimized is generality, then the solutions are given directly by either S or G (depending on whether one wishes to minimize or to maximize). If on the other hand one wishes to optimize w.r.t. frequency then one needs to compute (or remember) the frequency of all elements in either G or S. The answers to the queries are then those patterns within either G or S that are optimal.

5.2 Heuristic Solvers

So far, we have described *complete* solvers, which will find all solutions within the specified constraints. However, completeness often comes at a (computational) cost. Therefore, complete solvers may not always be desirable. There are at least two situations already encountered where this might be the case.

First, the provided primitives for optimisation were so far quite simple. Also, the criterion one may want to optimize is not necessarily frequency. Indeed, one is often more interested in accuracy, or entropy, etc. To optimize w.r.t. e.g. accuracy one cannot employ the above sketched method because accuracy involves combining maximum frequency on positives and minimum frequency on negatives. Hence, optimal patterns migh lie in the middle of the versionspace.

Second, there is a discrepancy between answering simple queries and answering queries over multiple inter-related patterns. There is good hope and evidence that the former solver is reasonably efficient, but it is also clear that the latter one is much less efficient, because it merely enumerates all possibilities.

Therefore, we need to extend the current solver with *heuristic* methods. This situation is again akin to what happens in constraint logic programming (cf. [16]). The effect of heuristic methods would be that queries would get a heuristic answer, that it might be that some solutions are missed, and also - in the case of optimizations - that suboptimal solutions might be generated. Of course, in such cases the user should be aware of this. When allowing for heuristic methods, it

would be possible to extend the database mining engine with various well-known database mining algorithms, e.g. with a beam-search procedure to greedily find the most interesting clauses in predictive modelling.

5.3 The Knowledge Discovery Cycle

Knowledge discovery in databases typically proceeds in an iterative manner. Data are selected, possibly cleaned, formatted, input in a data mining engine, and results are analysed and interpreted. As first results often can be improved, one would typically re-iterate this process untill some kind of a local optimum has been reached (cf. [10]).

Because knowledge discovery is an iterative process data mining tools should support this process. One consequence of the the iterative nature of knowledge discovery in our context is that many of the queries formulated to the database mining engine will be related. Indeed, one can imagine that various queries are similar except possibly for some parameters such as tresholds, data sets, pattern syntax, etc. The relationships among consecutive queries posed to the datamining engine should provide ample opportunities for optimization. The situation is - to some extent - akin to the way that views are dealt with in databases (cf. e.g. [9]). Views in databases are similar to patterns in data mining in that both constructs are virtual data structures, i.e. they do not physically exist in the database. Both forms of data can be queries and it is the task of the engines to efficiently answer questions concerning these virtual constructs.

Answering queries involving views can be realized in essentially in two different ways. First, one can *materialize* views, which means that one generates the tuples in the view relation explicitly, and then processes queries as if a normal relation was queried. Second, one can perform *query modification*, which means that any query to a view is "unfolded" into a query over the base relations. The advantage of materialization is that new queries are answered much faster whereas the disadvantage is that one needs to recompute or update the view whenever something changes in the underlying base relations. At a more general level, this corresponds to the typical computation versus storage trade-off in computer science.

These two techniques also apply to querying patterns in data mining. Indeed, if consecutive queries are inter-related, it would be useful to store the results (and possibly also the intermediate results) of one query in order to speed up the computation of the latter ones. This corresponds to materializing the patterns (together with accompanying information). Doing this would result in effective but fairly complicated solvers.

6 Related Work

RDM is related to other proposals for database mining query languages such as e.g. [17,11,2]. However, it differs from these proposals in a number of respects. First, due to the use of deductive databases as the underlying database model,

RDM allows to perform pattern discovery over multiple relations as in the field of inductive logic programming (cf. [5]). Secondly, a number of new and useful primitives are foreseen. Using RDM one is not restricted to finding frequent patterns, but one may also look for infrequent ones with regards to certain sets of (negative) examples. One can also require that certain examples are (resp. are not) covered by the patterns to be induced. Thirdly and most importantly, RDM allows to combine different primitives when searching for patterns. Finally, its embedding within PROLOG puts database mining on the same methodological grounds as constraint programming.

As another contribution, we have outlined an efficient algorithm for answering complex database mining queries. This algorithm integrates the principles of APRIORI with those of versionspaces and thus provide evidence that RDM can be executed efficiently. It also provides a generalized theoretical framework for datamining. The resulting framework is somewhat related to the levelwise techniques sketched by [14] who also link the levelwise algorithm to the S set of Mitchell's versionspace approach but do not further exploit the versionspace model as we do here. A prototype implementation of RDM along these lines already exists and might be put in the public domain soon.

There is plenty of further work that remains. First, a better implementation of the solver needs to be implemented. Secondly, the theoretical issues involved in answering complex database mining queries need to be addressed. Thirdly, the relationship with constraint logic programming needs to be examined further.

Finally, the author hopes that this work also provides a new perspective on inductive logic programming. The reason for this hope is that it puts inductive logic programming on the same methodological grounds as constraint logic programming. So far, inductive logic programming has mainly used the representations offered by computational logic but has basically employed the machine learning methodology. Exploiting the view of inductive logic programming as constraint processing one might bridge this gap.

Acknowledgements

This work was initially supported by the Fund for Scientific Research, Flanders, and by the ESPRIT IV project no 20237 on Inductive Logic Programming II. The author is grateful to Jean Francois Boulicaut, Maurice Bruynooghe and Luc Dehaspe for interesting discussions on this, to Hendrik Blockeel for comments on this paper, and especially to Imielinski and Mannila for inspiring this work through their CACM paper.

References

1. R. Agrawal, T. Imielinski, A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1993.

2. J-F. Boulicaut, M. Klemettinen, H. Mannila. Querying inductive databases: a case study on the MINE RULE operator. in: *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Artificial Intelligence, Vol. 1510, , Springer-Verlag, 1998.
in Proceedings of PKDD 98, Lecture Notes in AI, 1998.
3. I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990. 2nd Edition.
4. L. Dehaspe and L. De Raedt, WARMR : Wanted Association Rules over Multiple Relations, In *Proceedings of the International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 1297, Springer Verlag, 1997.
5. L. Dehaspe, H. Toivonen and R.D. King. Finding frequent substructures in chemical compounds, in *Proceedings of KDD-98*, 1998.
6. L. Dehaspe, H. Toivonen. Discovery of Frequent Datalog Patterns, in *Data Mining and Knowledge Discovery* , Vol. 3, 1999.
7. L. De Raedt, An inductive logic programming query language for database mining (Extended Abstract), in Calmet, J. and Plaza, J. (Eds.) *Proceedings of Artificial Intelligence and Symbolic Computation*, Lecture Notes in Artificial Intelligence, Vol. 1476, Springer Verlag, 1998.
8. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
9. Elmasri, R. and Navathe, S. Fundamentals of database systems. Benjamin Cummings, 2nd Edition, 1994.
10. Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. Advances in Knowledge Discovery, The MIT Press, 1996.
11. T. Imielinski and H. Mannila. A database perspectivce on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.
12. T. Imielinski, A. Virmani, and A. Abdulghani. A discovery board application programming interface and query language for database mining. In *Proceedings of KDD 96*. AAAI Press, 1996.
13. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
14. H. Mannila and H. Toivonen, Levelwise search and borders of theories in knowledge discovery, *Data Mining and Knowledge Discovery*, Vol. 1, 1997.
15. H. Mannila. Inductive databases. in *Proceedings of the International Logic Programming Symposium*, MIT Press, 1997.
16. Marriott, K. and Stuckey, P. J. Programming with constraints : an introduction. The MIT Press. 1998.
17. R. Meo, G. Psaila and S. Ceri, An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, Vol. 2, 1998.
18. C. Mellish. The description identification algorithm. *Artificial Intelligence*, Vol. 52, 1990.
19. T. Mitchell. Generalization as Search, *Artificial Intelligence*, Vol. 18, 1980.
20. S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
21. G. Plotkin, A note on inductive generalization, *Machine Intelligence*, Vol. 3, 1970.
22. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
23. W. Shen, K. Ong, B. Mitbander, and C. Zaniolo. Metaqueries for data mining. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 375–398. The MIT Press, 1996.

Induction of Recursive Theories in the Normal ILP Setting: Issues and Solutions

Floriana Esposito, Donato Malerba, and Francesca A. Lisi

Dipartimento di Informatica, Università degli Studi di Bari,
Via Orabona 4, I-70126 Bari, Italy
{esposito | malerba | lisi}@di.uniba.it

Abstract. Induction of recursive theories in the normal ILP setting is a complex task because of the non-monotonicity of the consistency property. In this paper we propose computational solutions to some relevant issues raised by the multiple predicate learning problem. A separate-and-parallel-conquer search strategy is adopted to interleave the learning of clauses supplying predicates with mutually recursive definitions. A novel generality order to be imposed to the search space of clauses is investigated in order to cope with recursion in a more suitable way. The consistency recovery is performed by reformulating the current theory and by applying a layering technique based on the collapsed dependency graph. The proposed approach has been implemented in the ILP system ATRE and tested in the specific context of the document understanding problem within the WISDOM project. Experimental results are discussed and future directions are drawn.

1 Introduction

Inductive learning of recursive logical theories is equivalent to learning multiple predicate definitions from a set of examples. De Raedt *et al.* [9] have showed that learning multiple predicates is more difficult than learning a single predicate. In fact, the former task is not limited to the generation of several *independent* predicate definitions, but involves the discovery of concept dependencies. A wrong hypothesis on concept dependencies may significantly affect the learning results. Moreover, the ordering typically used in inductive logic programming (ILP), namely □-subsumption [26], is not sufficient to guarantee the completeness and consistency of learned definitions with respect to logical entailment.

The main problems raised by multiple/recursive predicate learning can be explained in terms of an important property of the normal ILP problem setting: Whenever two individual clauses are consistent on the data, their conjunction need not to be consistent on the same data [11]. As a consequence, clauses supplying predicates with multiple/recursive definitions should not be learned individually but, in principle, they should be generated all together.

In order to overcome these problems, it has been proposed to work on a *weak* setting of ILP [14], in which the monotonicity property is satisfied: Whenever two individual clauses are valid on the data, their conjunction will also be valid on the

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 93-111, 2000.

data. In this setting, clauses can be investigated independently of each other since their interactions are no longer important [10]. On the other hand, the weak setting produces properties of examples instead of rules generating examples. This kind of hypotheses cannot always be used for predicting the truth values of facts. When we are interested in predictions, then the normal ILP setting is more appropriate.

Several studies on the problem of learning restricted forms of recursive theories in the normal ILP setting have been presented in the literature. Cohen [7] proves positive and negative results on the pac-learnability of several classes of logic theories that are allowed to include a recursive clause. Cameron-Jones and Quinlan [5] investigate a heuristic method for preventing infinite recursion in single predicate definitions with the system FOIL. De Raedt *et al.* [9] propose an algorithm, named MPL, that performs a greedy hill-climbing search for learning multiple predicate definitions. Giordana *et al.* [16] define a bottom-up learning algorithm, called RTL, that first learns a hierarchical (i.e., non-recursive) theory T which is complete and consistent, and then tries to synthesize a simple recursive theory from T. Aha *et al.* [1] have developed a system called CRUSTACEAN which is able to learn recursive definitions consisting of one unit clause and a two-literals recursive clause. Boström [3] proposes an algorithm that, under some assumptions, correctly specializes a given recursive theory with respect to positive and negative examples. Idestam-Almquist [17] suggests a technique to efficiently learn recursive definitions including one base clause and one tail recursive clause from a random sample of examples. An iterative bootstrap induction method for learning recursive predicate definitions has been studied by Jorge and Brazdil [18]. Finally, Mofizur and Numao [23] adopt a top-down approach to learning recursive programs with only one recursive clause. A thorough overview of achievements in the inductive synthesis of recursive logic programs can be found in [15].

In this paper, a new approach to the problem of learning multiple dependent concepts is proposed. It differs from De Raedt *et al.*'s approach in three aspects: the learning strategy, the generalization model, and the strategy to recover the consistency property of the learned theory when a new clause is added. These ideas have been implemented in the learning system ATRE [21] and tested in the specific context of the document understanding problem within the WISDOM project. In fact, the rules induced by ATRE are used by the document analysis and recognition system WISDOM++ [13] in order to recognize semantically relevant layout components (also called *logical components*) in documents being processed.

The paper is organized as follows. Section 2 introduces the issues related to the induction of recursive logical theories. Section 3 illustrates the learning strategy adopted by ATRE. Section 4 is devoted to the generalization model whose implementation is also sketched. A solution to the problem of recovering non-monotonic theories is proposed in Section 5. In Section 6 the application of ATRE to the document understanding problem and experimental results obtained on a set of real-world multi-page documents are described. Finally, in Section 7 our conclusions are drawn.

2 Recursive Theories: Learning Issues

Henceforth, the term *logical theory* will denote a set of definite clauses. Every logical theory T can be associated with a directed graph $\Gamma(T)=<N,E>$, called the *dependency graph* of T, in which (i) each predicate of T is a node in N and (ii) there is an arc in E directed from a node a to a node b iff there exists a clause C in T such that a and b are the predicates of a positive literal occurring in the head and in the body of C, respectively.

A dependency graph allows to represent the predicate dependencies of T, where a *predicate dependency* is defined as follows:

Definition 1 (predicate dependency [8]). A predicate p depends on a predicate q in a theory T iff (i) there exists a clause C for p in T such that q occurs in the body of C; or (ii) there exists a clause C for p in T with some predicate r in the body of C that depends on q.

It is straightforward to notice that the direct (i) and indirect (ii) predicate dependencies of T are represented as arcs and paths respectively in $\Gamma(T)$. The correspondence may be highlighted by reformulating the problem from an algebraic point of view. Let $\Gamma(T)$ be the set of predicates occurring in the logical theory T. The direct (i) predicate dependencies in T may be mathematically depicted as instances of a binary relation on $\Gamma(T)$, namely $R_{dpd} \subseteq \Gamma(T) \times \Gamma(T)$. The binary relation R_{pd} for predicate dependencies is the transitive closure of R_{dpd}. Given that each binary relation can be associated with a directed graph, the graph corresponding to R_{dpd} is just the dependency graph $\Gamma(T)=<\Gamma(T),R_{dpd}>$.

Definition 2 (recursive theory). A logical theory T is *recursive* if the dependency graph $\Gamma(T)$ contains at least one cycle.

In *simple* recursive theories all cycles in the dependency graph go from a predicate p into p itself, that is simple recursive theories may contain recursive clauses, but cannot express mutual recursion. An example of dependency graph for a recursive theory is given in Fig. 1.

odd(succ(X)) ← even(X)
even(succ(X)) ← odd(X)
even(X) ← zero(X)

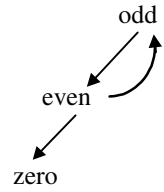


Fig. 1. A recursive theory and its corresponding dependency graph for the predicates *even* and *odd*.

Most studies on the problem of induction of recursive theories have been concentrated on learning a simple recursive predicate definition [5, 17, 18, 23]. In this case, the main issue is how to guarantee that learned definitions are intensionally complete and consistent. Learning simple recursive theories is more complicated, since it is necessary to discover the right order in which predicates should be learned [16], that is the dependency graph of the theory. Once such order has been

determined, possibly using statistical techniques, the problem can be boiled down to learning single predicate definitions [20]. The learning problem becomes harder for recursive theories, because the learning of one (possibly recursive) predicate definition should be *interleaved* with the learning of the other ones. One way to build such interleaving is by parallel learning clauses for different predicates. This is, indeed, the strategy adopted by ATRE.

3 The Learning Strategy

The high-level learning algorithm in ATRE belongs to the family of *sequential covering* (or *separate-and-conquer*) algorithms [22] since it is based on the strategy of learning one clause at a time (procedure LEARN-ONE-RULE), removing the covered examples and iterating the process on the remaining examples. Indeed, a recursive theory T is built step by step, starting from an empty theory T_0, and adding a new clause at each step. In this way we get a sequence of theories

$$T_0 = \square, T_1, \square, T_i, T_{i+1}, \square, T_n = T$$

such that $T_{i+1} = T_i \square \{C\}$ for some clause C, and all theories in the sequence are consistent with respect to the training set. If we denote with $LHM(T_i)$ the least Herbrand model of a theory T_i, the stepwise construction of theories entails that $LHM(T_i) \square LHM(T_{i+1})$, for each $i \square \{0, 1, \square, n\text{-}1\}$. Indeed the addition of a clause to a theory can only augment the least Herbrand model of the theory. Henceforth we will assume that both positive and negative examples of predicates to be learned are represented as ground atoms with a **+** or **-** label. Therefore examples may or may not be elements of the Herbrand models $LHM(T_i)$.

Let $pos(LHM(T_i))$ and $neg(LHM(T_i))$ be the number of positive and negative examples in $LHM(T_i)$, respectively. If we guarantee that $pos(LHM(T_i)) < pos(LHM(T_{i+1}))$ for each $i \square \{0, 1, \square, n\text{-}1\}$, and that $neg(LHM(T_i))=0$ for each $i \square \{0, 1, \square, n\}$, then after a finite number of steps a theory T, which is complete and consistent, is built. Whether the theory T is "correct", that is whether it classifies correctly all other examples *not* in the training set, cannot be established, since no information on the generalization accuracy can be drawn from the same training data. In fact, the selection of the "best" theory is always made on the basis of an inductive bias embedded in some heuristic function or explicitly expressed by the user of the learning system (preference criterion).

In order to guarantee the first condition above, namely $pos(LHM(T_i)) < pos(LHM(T_{i+1}))$, we suggest to proceed as follows. First, a positive example e^+ of a predicate p to be learned is selected, such that e^+ is not in $LHM(T_i)$. The example e^+ is called *seed*. Then the space of definite clauses more general than e^+ is explored, looking for a clause C, if any, such that $neg(LHM(T_i \square \{C\}))=0$. In this way we guarantee that the second condition above holds as well. When found, C is added to T_i giving T_{i+1}. If some positive examples are not included in $LHM(T_{i+1})$ then a new seed is selected and the process is repeated.

The most relevant novelties of the learning strategy sketched above are embedded in the design of the procedure LEARN-ONE-RULE being proposed. Indeed, it implements a parallel general-to-specific example-driven search strategy in the space of definite clauses whose ordering (called *generalized implication*) is explained in

Section 4. The search space is actually a forest of as many search-trees (called *specialization hierarchies*) as the number of chosen seeds, where at least one seed per incomplete predicate definition is kept. Each search-tree is rooted with a unit clause and ordered by generalized implication. The forest can be processed in parallel by as many concurrent tasks as the number of search-trees. Each task traverses the specialization hierarchies top-down (or general-to-specific), but synchronizes traversal with the other tasks at each level. Initially, some clauses at depth one in the forest are examined concurrently. Each task is actually free to adopt its own search strategy, and to decide which clauses are worth to be tested. If none of the tested clauses is consistent, clauses at depth two are considered. Search proceeds towards deeper and deeper levels of the specialization hierarchies until at least one consistent clause is found. Task synchronization is performed after that all "relevant" clauses at the same depth have been examined. A supervisor task decides whether the search should carry on or not on the basis of the results returned by the concurrent tasks. When the search is stopped, the supervisor selects the "best" consistent clause according to the user's preference criterion. This strategy has the advantage that simpler consistent clauses are found first, independently of the predicates to be learned. Moreover, the synchronization allows tasks to save much computational effort when the distribution of consistent clauses in the levels of the different search-trees is uneven. The parallel exploration of the specialization hierarchies for the concepts *even* and *odd* is shown in Fig. 2.

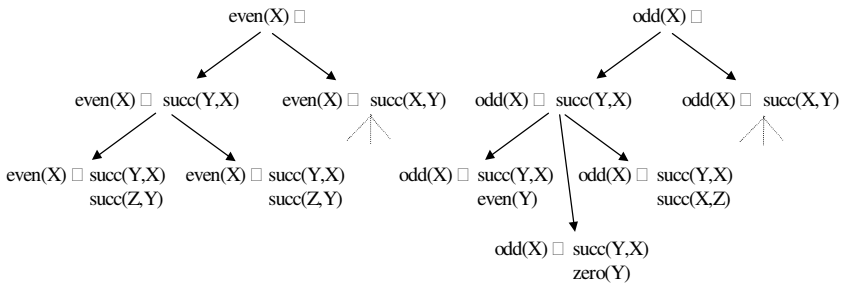


Fig. 2. Parallel search for the concepts *even* and *odd*.

To sum up, this *separate-and-parallel-conquer* search strategy provides us with a solution to the problem of interleaving the induction process for distinct predicate definitions. A different approach has been followed in MPL [9], which performs a greedy hill-climbing search for theories and a beam search for each single clause. Clauses can be generated by means of two types of refinement operators, one for the body and one for the head. In particular, it is possible to generate the body of a clause without specifying its head. The generation of clauses like $p \leftarrow p$ is not forbidden, and it is possible to learn a recursive clause for a predicate p, before that a base clause for p has been found. In fact, the algorithm uses a depth-bounded interpreter for checking whether an induced theory logically entails an example.

4 The Generalization Model

A more precise definition of the search space of the LEARN-ONE-RULE stage is necessary. A generality order (or *generalization model*) provides a basis for organizing this search space. It can be proven that Buntine's *generalized subsumption* [4] is an order suitable for *simple* recursive theories, since it is neither too strong nor too weak.[1]

Definition 3 (generalized subsumption). Let C and D be two definite clauses with disjoint variables:

C: $C_0 \leftarrow C_1, C_2, \ldots, C_n$

D: $D_0 \leftarrow D_1, D_2, \ldots, D_m$

Then C is *more general than D* under generalized subsumption with respect to a theory T, denoted $C \geq_T D$, if there exists a substitution θ such that $C_0\theta = D_0$ and for each substitution σ that grounds the variables in D using new constants not occurring in C, D, and T, it happens that:

$$T \cup \{D_1\theta\sigma, D_2\theta\sigma, \ldots, D_m\theta\sigma\} \vdash_{SLD} (C_1, C_2, \ldots, C_n)\theta\sigma.$$

Unfortunately generalized subsumption is too weak for recursive theories. Thus, we may resort to Plotkin's notion of *relative generalization* [26, 27].

Definition 4 (relative generalization). Let C and D be two definite clauses. Then C is *more general than D* under relative generalization with respect to a theory T if there exists a substitution θ such that $T \models \forall(C\theta \rightarrow D)$.

However, relative generalization is too strong for our goals. Taken literally, it would lead us to consider unintuitive solutions of the conquer stage as illustrated in the following example.

Example Let us consider the following examples:

+: $\{p(a), p(b)\}$

—: $\{p(c)\}$

the following background knowledge:

BK: $\{q(a), r(a,b), s(b), r(c,b)\}$

and the following incomplete theory built at the first step:

T_1: $p(X) \leftarrow q(X)$

Let $p(b)$ be the selected seed. A desirable search space of clauses more general than $p(b)$ given $BK \cup T_1$ would be the set of clauses whose head is $p(X)$, since our aim is to induce a predicate definition for p. This space contains, for instance, the clause $p(X) \leftarrow s(X)$. However, the space of clauses that are relatively more general than $p(b)$ given $BK \cup T_1$ includes other solutions, such as $q(Y) \leftarrow s(Y)$. This last clause is correct and even relatively more general than $p(X) \leftarrow s(X)$. Nevertheless it is unacceptable from an intuitive point of view because it seems not to be related to the target predicate p. In this work, we do not consider these less intuitive solutions.

This restriction is reflected by an ordering, named *generalized implication*, which is a special case of relative generalization.

---

[1] Informally, an order is too strong for a class L of theories when it can be used to organize theories of a strictly wider class $L' \supset L$ according to logical entailment. If the organization of theories in L is not consistent with logical entailment, then the order is too weak.

Definition 5 (generalized implication). Let C and D be two definite clauses. Then C is *more general than* D under generalized implication with respect to a theory T, denoted as $C \preceq_{T,\theta} D$, if there exists a substitution θ such that $head(C)\theta = head(D)$ and $T \models \forall (C\theta \rightarrow D)$.

This generality order can be proved to be strictly weaker than relative generalization and strictly stronger than generalized subsumption. Moreover, the following properties hold.

Proposition 1. *Let C, D and E be definite clauses and T a theory. The generalized implication order satisfy the properties of:*
i) reflexivity: $C \preceq_{T,\theta} C$
ii) transitivity: $C \preceq_{T,\theta} D$ *and* $D \preceq_{T,\theta} E$ *then* $C \preceq_{T,\theta} E$

Proof i) Trivial if the empty substitution is chosen. ii) By definition, there exists a substitution θ_1 such that $head(C)\theta_1 = head(D)$ and $T \models \forall (C\theta_1 \rightarrow D)$ and there exists a substitution θ_2 such that $head(D)\theta_2 = head(E)$ and $T \models \forall (D\theta_2 \rightarrow E)$. Let $\theta = \theta_1\theta_2$ be the substitution obtained by composition of θ_1 and θ_2. Let us prove that θ is a substitution such that C is more general than E under generalized implication. By definition of compound substitution, we can say that $head(C)\theta = head(C)\theta_1\theta_2 = head(D)\theta_2 = head(E)$. Given that the implication is monotone with respect to the application of a substitution, $T \models \forall (C\theta_1 \rightarrow D)$ entails $T \models \forall (C\theta_1\theta_2 \rightarrow D\theta_2)$. From $T \models \forall (C\theta_1\theta_2 \rightarrow D\theta_2)$ rewritten as $T \models \forall (C\theta \rightarrow D\theta_2)$ and $T \models \forall (D\theta_2 \rightarrow E)$, $T \models \forall (C\theta \rightarrow E)$ follows.

It can also be proved the semi-decidability of the generalized implication, namely the termination of the generalized implication test is guaranteed when $C \preceq_{T,\theta} D$. However, such a negative result is overcome when Datalog clauses [6] are considered. In fact, the restriction to function-free clauses is common in ILP systems, which remove function symbols from clauses and put them in the background knowledge by techniques such as flattening [29].

The generalization model represents another difference between ATRE and MPL, whose refinement operators are based on θ-subsumption. MPL adopts two different generalization models during its search: θ-subsumption while learning a single clause, and logical entailment while learning the whole theory. Therefore, two distinct checks are performed by the system for each learned clause: a local consistency/completeness check based on θ-subsumption and a global check based on logical entailment.

4.1 An Implementation of the Generalized Implication Test

A naive implementation of the generalized implication test is obtained by computing the least Herbrand models of $\{C\theta\} \cup T$ and $\{D\} \cup T$: $C \preceq_{T,\theta} D$ if and only if $LHM(\{D\} \cup T) \supseteq LHM(\{C\theta\} \cup T)$ for some θ such that $head(C)\theta = head(D)$. Since the least Herbrand model of a theory T coincides with the least fixed point of the immediate consequence operator T_P[2] we have an operative way to compute the least

[2] The standard notation used in logic programming and deductive databases for the immediate consequence operator is T_P, where P is the logic program or the Datalog program.

Herbrand models. Moreover, the convergence to the fixpoint is guaranteed by the finiteness of the Herbrand base for Datalog theories.

One reason for inefficiency in the naive evaluation is that ground facts in $LHM(T)$ may be computed many times during the iterative application of Γ_T. Semi-naive evaluation partially overcomes this redundancy by partitioning T into *n layers* such that $T=T^0 \cup \ldots \cup T^i \cup \ldots \cup T^{n-1}$ and $LHM(T)= LHM(LHM(\cup_{j=0,\ldots,n-2} T^j) \cup T^{n-1})$.

It is worthwhile to notice that the computation of $LHM(LHM(\cup_{j=0,\ldots,i-1} T^j) \cup T^i)$ is equivalent to the iterative application of the immediate consequence operator to T^i starting from the interpretation $LHM(\cup_{j=0,\ldots,i-1} T^j)$, that is $\Gamma_{T^i} (LHM(\cup_{j=0,\ldots,i-1} T^j))$. In this way, clauses in $T^0 \cup \ldots \cup T^{i-1}$ are not considered anymore while computing the logical consequences of T^i.

Example Let T be the following theory:

C_1: $p(X) \leftarrow q(X)$

C_2: $q(Y) \leftarrow r(X,Y)$

and

BK: $\{q(a), r(a,b), s(b), r(c,b)\}$

Let us suppose that the theory $T=BK \cup T$ may be partitioned as follows: $T=T^0 \cup T^1 \cup T^2$ where $T^0=\{r(a,b), s(b), r(c,b)\}$, $T^1=\{q(a), C_2\}$ and $T^2=\{C_1\}$.

Indeed, $LHM(T)= \{q(a), r(a,b), s(b), r(c,b), q(b), p(a), p(b)\}$, and

$LHM(T^0)= T^0$

$LHM(T^0 \cup T^1)=LHM(LHM(T^0) \cup T^1)=LHM(T^0 \cup T^1)=$

$= \{r(a,b),s(b),r(c,b),q(a),q(b)\}= \Gamma_{T^1} (\{r(a,b), s(b), r(c,b)\})=\Gamma_{T^1} (LHM(T^0))$

$LHM(T^0 \cup T^1 \cup T^2)=LHM(LHM(T^0 \cup T^1) \cup T^2)=LHM(\{r(a,b),s(b),r(c,b),q(a),q(b)\} \cup \{C_1\}=$

$=\{r(a,b),s(b),r(c,b),q(a),q(b),p(a),p(b)\}=$

$=\Gamma_{T^2} (\{r(a,b),s(b),r(c,b),q(a),q(b)\})=\Gamma_{T^2} (LHM(T^0 \cup T^1))$

Notice that according to the classical iterative application of the immediate consequence operator the ground atoms $p(a)$ and $q(b)$ would be computed both in $\Gamma_T \uparrow 2$ and $\Gamma_T \uparrow 3$, since:

$\Gamma_T \uparrow 0 := \emptyset$

$\Gamma_T \uparrow 1 := \Gamma_T(\Gamma_T \uparrow 0) = BK$

$\Gamma_T \uparrow 2 := \Gamma_T(\Gamma_T \uparrow 1) = BK \cup \{p(a), q(b)\}$

$\Gamma_T \uparrow 3 := \Gamma_T(\Gamma_T \uparrow 2) = BK \cup \{p(a), q(b)\} \cup \{ p(a), q(b), p(b)\}$

Issues related to the problem of finding layers of a recursive theory T such that $LHM(T)=LHM(\cup_{j=0,\ldots,n-1} T^j)=LHM(LHM(\cup_{j=0,\ldots,n-2} T^j) \cup T^{n-1})$ are to be addressed. Difficulties arise because the dependency graph $\delta(T)$ is a directed cyclic graph. In order to remove cycles from $\delta(T)$ we resort to the notion of *strongly connected component* of a directed graph [19].

Definition 6 (collapsed dependency graph). Let $\delta(T)$ be the dependency graph of a logical theory T. The *collapsed dependency graph* of T, denoted as $\hat{\delta}(T)$, is a

---

Henceforth, the operator will be denoted by Γ_T in order to avoid confusion with the symbol T used for logical theories.

directed acyclic graph (dag) obtained by collapsing each (maximal) strongly connected component of $\Gamma(T)$ into a single node.

Given the properties of *dag*'s, it is easy to compute the level of a predicate $p \in \Gamma(T)$ as the maximum distance of $[p]$ from a terminal node in $\hat{\Gamma}(T)$ where terminal nodes are nodes with no out-coming edges.

Definition 7 (predicate level). Let $\hat{\Gamma}(T)$ be the collapsed dependency graph of a logical theory T. The level of a predicate $p \in \Gamma(T)$ is given by:

$level(p) = 0$ if $[p]$ is a terminal node in $\hat{\Gamma}(T)$,

$1 + \max \{level(q) \mid q \in \Gamma(T)$ and $[q]$ is a child of $[p]$ in $\hat{\Gamma}(T)\}$ otherwise

Any logical theory can be layered on the basis of the level of its predicates.

Definition 8 (layered theory). Let $\hat{\Gamma}(T)$ be the collapsed dependency graph of a logical theory T. Then T can be partitioned into n disjoint sets of clauses $T=T^0 \cup \ldots \cup T^i \cup \ldots \cup T^{n-1}$, called *layers,* such that

$\forall i \in 0, .., n-1: \Gamma(T^i) = \{p \in \Gamma(T) \mid level(p)=i\}$.

It is worthwhile to observe that such technique for the layering of a logical theory induces a total order on the layers, $T^0 \prec \ldots \prec T^i \prec \ldots \prec T^{n-1}$.

Example Let T be a theory obtained by the union of a background knowledge

BK: $\{f(a), s(a,b), s(b,c), s(c,d), s(d,e)\}$

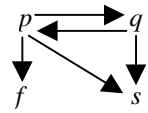
and a theory T consisting of

C_1: $p(X) \leftarrow f(X)$

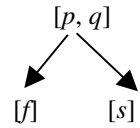
C_2: $q(Y) \leftarrow p(Z), s(Z,Y)$

C_3: $p(U) \leftarrow q(V), s(V,U)$

Given $T= BK \cup T$, then $\Gamma(T)=\{f, s, p, q\}$ while $\hat{\Gamma}(T)$ is the following:



Thus $\hat{\Gamma}(T) = \{[f], [s], [p, q]\}$, and $\hat{\Gamma}(T)$ is the following graph:



from which the two layers $T^0=BK$ and $T^1=T$ are extracted.

The following proposition can be proved.

Proposition 2. *Let H be a ground atom with predicate symbol $p \in \Gamma(T^k)$. Then $H \in LHM(T)$ if and only if $H \in LHM(LHM(\cup_{r=1,\ldots,k-1} T^r) \cup T^k)$.*

Proof

($\Rightarrow$) By induction over the layer k.

k=0. Let $H \in LHM(T)$. Then by the fix-point theorem, $\exists i>0$ such that $H \in T_T \uparrow i$, that is, by definition of immediate consequence operator, there exists a ground clause of T, $H \leftarrow A_1,\ldots,A_m \in ground(T)$, where $A_j \in T_T \uparrow (i-1)$, j=1,2,…,m. Since $p \in \Gamma(T^0)$ then

$H \square A_1,\ldots,A_m \square ground(T^0)$. We want to prove that each $A_j \square LHM(T^0)$, from which $H \square LHM(T^0)$ follows.

The proof continues by induction over the iteration step i.

i=1. Since $H \square\square_T \square 1$ then $H \square T$, that is H is a ground fact of the theory T. More specifically, $H \square T^0$, therefore H is in the $LHM(T^0)$.

i>1. Each $A_j \square\square_T \square(i\text{-}1)$, therefore $A_j \square$LHM(T). From the construction of the layers follows that each A_j is a ground atom with predicate symbol $p_j \square\square(T^0)$. By the induction hypothesis each $A_j \square LHM(T^0)$, j=1,2, …, m.

k>0. Again, let $H \square LHM(T)$. Then for the fix-point theorem, $\square i$>0 such that $H \square\square_T \square i$, that is, by definition of immediate consequence operator, there exists a ground clause of T, $H \square A_1,\ldots,A_m \square ground(T)$, where $A_j \square\square_T \square(i\text{-}1)$, j=1, 2, …,m. We want to prove that $A_j \square LHM(LHM(\square_{r=1,\ldots,k-1} T^r) \square T^k)$, from which $H \square LHM(LHM(\square_{r=1,\ldots,k-1} T^r) \square T^k)$ follows.

The proof continues by induction over the step i.

i=1. Since $H \square\square_T \square 1$ then $H \square T$, that is H is a ground fact of the theory T. More specifically, $H \square T^k$, therefore H is in the $LHM(T^k)$, and more in general H is in $LHM(LHM(\square_{i=1,\ldots,k-1} T^r) \square T^k)$, since the addition of a set of clauses (i.e., the ground clauses $LHM(\square_{i=1,\ldots,k-1} T^i)$) to a theory T^k can only augment the least Herbrand model of the theory.

i>1. From the construction of the layers follows that each A_j is a ground atom with predicate symbol $p_j \square\square(T^r)$, with $r \square k$. Moreover $A_j \square\square_T \square(i\text{-}1)$, therefore $A_j \square LHM(T)$. By both inductive hypotheses (over k and i), we may say that $A_j \square LHM(LHM(\square_{r=1,\ldots,k-1} T^r) \square T^k)$, for each j=1, 2, …,m, and then conclude that $H \square LHM(LHM(\square_{r=1,\ldots,k-1} T^r) \square T^k)$.

($\square$) Let $H \square LHM(LHM(\square_{r=1,\ldots,k-1} T^r) \square T^k)$. In particular, if $H \square LHM(\square_{r=1,\ldots,k-1} T^r)$ then $H \square LHM(T)$, since $\square_{r=1,\ldots,k-1} T^r \square T$. Otherwise, H has been obtained by applying iteratively the immediate consequence operator $\square_{T^k}$ starting from the interpretation $LHM(\square_{r=1,\ldots,k-1} T^r) \square LHM(T)$, that is $H \square \square_{T^k}$ (LHM($\square_{r=1,\ldots,k-1} T^r$))$\square\square$. Since $\square_{T^k}$ is monotone and $T^k \square T$, $\square_{T^k}$ ($LHM(\square_{r=1,\ldots,k-1} T^r)$)$\square\square$ $\square$ $\square_{T^k}$ ($LHM(T)$)$\square\square$=$LHM(T)$. Therefore $H \square LHM(T)$. ■

This proposition states that a necessary and sufficient condition for a ground atom with predicate symbol $p \square \square(T^k)$ being in $LHM(T)$ is that H is computed by iteratively applying the immediate consequence operator $\square_{T^k}$ starting from the interpretation $LHM(\square_{r=1,\ldots,k-1} T^r)$. Proposition 2 can be used to prove the following theorem on the least Herbrand model of a layered theory.

Theorem. *Let T be a theory which has been partitioned into n layers according to the criterion defined in Definition 8. Then*

$$\square\, n \square 1\text{: } LHM(T)=LHM(\square_{r=0,\ldots,n-1} T^r)=LHM(LHM(\square_{r=0,\ldots,n-2} T^r) \square T^{n-1}).$$

Proof

($\square$) Let P^0, P^1,$\square$, P^{n-1} be a partition of $LHM(T)$ such that each P^i, i=0, $\square$, n-1, contains only ground atoms with some predicate symbol in $\square(T^i)$. From Proposition 2 it follows $P^i \square LHM(LHM(\square_{r=1,\ldots,i-1} T^r) \square T^i)$ and $LHM(T)=P^0 \square P^1 \square\square$ $P^{n-1} \square LHM(T^0) \square$

$LHM(LHM(T^0) \cup T^1) \cup \ldots \cup LHM(LHM(\cup_{r=0,\ldots,n-2} T^r) \cup T^{n-1}) = LHM(LHM(\cup_{r=0,\ldots,n-2} T^r) \cup T^{n-1})$ since $LHM(\cup_{r=0,\ldots,i-1} T^r) \cup LHM(LHM(\cup_{r=0,\ldots,i-1} T^r) \cup T^i)$.

($\supseteq$) Let H be in $LHM(LHM(\cup_{r=0,\ldots,n-2} T^r) \cup T^{n-1})$. If $H \in LHM(\cup_{r=0,\ldots,n-2} T^r)$ then $H \in LHM(T)$, since $\cup_{r=0,\ldots,n-2} T^r \subseteq T$. Otherwise, H has been obtained by applying iteratively the immediate consequence operator $T_{P \cup BK}^{n-1}$ starting from the interpretation $LHM(\cup_{r=1,\ldots,n-2} T^r) \cup LHM(T)$, that is $H \in T_{P \cup BK}^{n-1}(LHM(\cup_{r=1,\ldots,n-2} T^r)) \cup \ldots$. Since $T_{P \cup BK}^{n-1}$ is monotone and $T^{n-1} \subseteq T$, $T_{P \cup BK}^{n-1}(LHM(\cup_{r=1,\ldots,n-2} T^r)) \cup \ldots \cup T_{P \cup BK}^{n-1}(LHM(T)) \cup \ldots = LHM(T)$. Therefore $H \in LHM(T)$. ■

To sum up, the layering of a theory provides a semi-naive way of computing the generalized implication test presented above. The importance of layering will be more evident when the problem of recovering consistency will be faced (see the following Section).

5 The Consistency Recovery Strategy

Another learning issue to be considered in the multiple predicate learning is the non-monotonicity of the normal ILP setting: Whenever two individual clauses are consistent on the data, their conjunction needs not to be consistent on the same data [11]. Algorithmic implications of this property may be effectively illustrated by means of an example.

Example. Let the following sets be positive examples, negative examples and background knowledge respectively:

+: $\{p(a),\ p(c), p(e), q(b)\}$
—: $\{q(d)\}$
BK: $\{f(a), s(a,b), s(b,c), s(c,d), s(d,e)\}$

Let us suppose that the following consistent, but not complete, recursive theory T_2 has been learned after two conquer stages:

C_1: $p(X) \leftarrow f(X)$
C_2: $q(Y) \leftarrow p(Z), s(Z,Y)$

Note that $C_1 \in \{C2\} \cup_{BK} \{p(a)\}$, and $C_2 \in \{C1\} \cup_{BK} \{q(b)\}$, that is T_2 entails $p(a)$ and $q(b)$ given *BK*. Since T_2 is incomplete, the learner will generate a new clause, say

C: $p(U) \leftarrow q(V), s(V,U)$

which is consistent (it covers $p(c)$ alone, given $T_2 \cup BK$), but when added to the recursive theory, it makes clause C_2 inconsistent ($C_2 \in \{C1,C\} \cup_{BK} \{q(d)\}$).

There are several ways to remove such inconsistency by revising the learned theory. Nienhuys-Cheng and de Wolf [25] describe a complete method to specialize a logic theory with respect to sets of positive and negative examples. The method is based upon unfolding, clause deletion and subsumption. These operations are not applied to the last clause added to the theory, but may involve any clause of the inconsistent theory. As a result, clauses learned in the first inductive steps could be totally changed or even removed. This theory revision approach, however, is not coherent with the stepwise construction of the theory T presented in Section 3, since it

re-opens the whole question on the validity of clauses added in the previous steps. An alternative approach consists of simple syntactic changes of the theory, whose ultimate effect is that of creating new *layers* in a logical theory, just as the stratification of a normal program creates new strata [2].

Our proposal of recovery strategy fits the latter approach since it is based on the layering technique illustrated in Section 4.1 (see Definition 8).

Example In the previous example, it is possible to define only two layers for T_2 BK
1. $T^0=BK$ with $\Box(T^0)=\{f, s\}$, and
2. $T^1=\{C_1, C_2, C\}$ with $\Box(T^1)=\{p, q\}$.

By reformulating C_1 and C_2 as follows:

C'_1: $p'(X) \leftarrow f(X)$

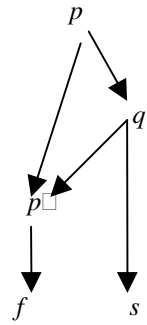
C'_2: $q(Y) \leftarrow p'(Z), s(Z,Y)$

and by adding the following two clauses:

C_3: $p(W) \leftarrow p'(W)$

C: $p(U) \leftarrow q(V), s(V,U)$

the new theory T'_2 will present four layers:
1. $T^0=BK$ with $\Box(T^0)=\{f, s\}$,
2. $T^1=\{C'_1\}$ with $\Box(T^1)=\{p'\}$,
3. $T^2=\{C'_2\}$ with $\Box(T^2)=\{q\}$, and
4. $T^3=\{C_3, C\}$ with $\Box(T^3)=\{p\}$.



It is straightforward to notice that the theory T'_2 is consistent. As effect of theory restructuring, p and q are no longer in the same equivalence class in the collapsed dependency graph, and the number of layers increased.

More generally, let $T=T^0 \cup \ldots \cup T^i \cup \ldots \cup T^{n-1}$ and suppose that the addition of a clause C to the theory T makes a clause in T^i inconsistent. The recovery strategy based on layering simply substitutes all occurrences in T of the predicate p in $head(C)$ with a new predicate symbol, p', before adding the two clauses C and $p(t_1,\ldots,t_n) \leftarrow p'(t_1,\ldots,t_n)$.

Proposition 3. *The new theory T' obtained as above has a number of layers greater than or equal to T.*

Proof. Let l be the $level(p)$ in $\hat{\Box}(T)$. Since p' replaces p in T it has the same level of p before theory restructuring, namely $level(p')=l$ in $\hat{\Box}(T')$. Moreover $level(p)>level(p')$ in $\hat{\Box}(T')$. If l equals the maximum level of a node in $\hat{\Box}(T)$, then the new theory T' has a predicate at a greater level, that is the number of layers in T' increases. Conversely, the level of all predicates q depending on p in T can either increase because of the breaking of equivalence classes or remain stable. ■

This proposition generalizes the consideration on the increase of layers made for the example above. The main problem remains the coverage of the new theory T'.

Proposition 4. *The new theory T' is consistent.*

Proof. Let $T' = T' \setminus \{C\}$. From the construction of T' follows $LHM(T') = LHM(T) \cup \{p'(t_1,\ldots,t_n) \mid p(t_1, \ldots,t_n) \in LHM(T)\}$. Indeed, T' simply renames p with p' and adds the clause $p(t_1, \ldots,t_n) \leftarrow p'(t_1, \ldots,t_n)$. T' is consistent with respect to the training set, since $LHM(T)$ does not include negative examples (T is consistent before adding

C), and no conflict can be generated between the new ground atoms $p(t_1,\ldots,t_n)$ and the training set. Moreover, no clause in T' depends on p, because of renaming. Therefore, $LHM(T') = LHM(LHM(T) \cup \{C\})$. Suppose that T' is inconsistent, that is there exists a ground atom $H \in LHM(T')$ such that H is a negative example in the training set. Obviously, $H \in LHM(T)$ since we have just proved T' being consistent. By hypothesis, the clause C is consistent given T (otherwise it would not be selected during the learning process), therefore $LHM(LHM(T) \cup \{C\})$ does not contain negative examples. Since $LHM(T') = LHM(T) \cup \{p(t_1,\ldots,t_n) \mid p(t_1, \ldots,t_n) \in LHM(T)\}$ and $p(t_1,\ldots,t_n)$ do not affect the set of consequences computed by the immediate consequence operator, we conclude that also $LHM(LHM(T') \cup \{C\})$ does not contain negative examples. ■

Proposition 5. $LHM(T) \subseteq LHM(T')$.

Proof. Let $T' = T \setminus \{C\}$. By construction of T' and T, the thesis follows from the chaining of the inclusions $LHM(T') = LHM(LHM(T') \cup \{C\}) \subseteq LHM(T)$ and $LHM(T') = LHM(T) \cup \{p(t_1,\ldots,t_n) \mid p(t_1,\ldots,t_n) \in LHM(T)\} \subseteq LHM(T)$. ■

To sum up, the new theory T' is consistent and keeps the original coverage of T.

It is noteworthy that in the proposed approach to consistency recovery new predicates are invented, aiming at accommodating previously acquired knowledge (theory) with the currently generated hypothesis (clause). This approach differs from that adopted by MPL, which recovers inconsistency by allowing removal of globally inconsistent clauses from the current theory.

6 Application to the Document Understanding Problem

The proposed approach to multiple predicate learning has been implemented in the learning system ATRE, whose representation language can be easily transformed into Datalog clauses extended with built-in predicates. An application to the problem of learning the mutual recursive predicate definition of *even* and *odd* is reported in [21]. ATRE has been also applied to the problem of processing printed documents and its induced logical theories are used by an intelligent document processing system, named WISDOM++ (http://www.di.uniba.it/~malerba/wisdom++/) [13]. Henceforth, only the specific problem of learning rules for document understanding will be addressed. The main novelty with respect to previous work is the automated discover of possible concept dependencies as well as the consideration of multi-page documents.

A document is characterized by two different structures representing both its internal organization and its content: The *layout* (or *geometrical*) structure and the *logical* structure. The former associates the content of a document with a hierarchy of layout objects, such as text lines, vertical/horizontal lines, graphic/photographic elements, pages, and so on. The latter associates the content of a document with a hierarchy of logical objects, such as sender/receiver of a business letter, title/authors of an article, and so on. Here, the term document understanding denotes the process of mapping the layout structure of a document into the corresponding logical structure. The document understanding process is based on the assumption that documents can be understood by means of their layout structures alone.

The mapping of the layout structure into the logical structure can be represented as a set of rules. Traditionally, such rules have been hand-coded for particular kinds of document [24], requiring much human tune and effort. We propose the application of inductive learning techniques in order to generate automatically the rules from a set of training examples. The user-trainer is asked to label some layout components of a set of training documents according to their logical meaning. Those layout components with no clear logical meaning are not labeled. Therefore, each document generates as many training examples as the number of layout components. Classes of training examples correspond to the distinct logical components to be recognized in a document. The unlabelled layout objects play the role of counterexamples for all the classes to be learned.

Each training example is represented as an *object* in ATRE, namely a multiple-head ground clause, where different constants represent distinct layout components of a page layout. The description of a document page is reported in Fig. 3 while Table 1 lists all descriptors used to represent a page layout of a multi-page document.

Table 1. Descriptors used by WISDOM++ to represent a page layout of a multi-page document.

Descriptor	***Domain***
page(page)	*Nominal domain:* first, intermediate, last_but_one, last
width(block)	*Integer domain:* (1..640)
height(block)	*Integer domain:* (1..875)
x_pos_centre(block)	*Integer domain:* (1..640)
y_pos_centre(block)	*Integer domain:* (1..875)
type_of(block)	*Nominal domain:* text, hor_line, image, ver_line, graphic, mixed
part_of(page,block)	*Boolean domain:* true if page contains block
on_top(block1,block2)	*Boolean domain:* true if block1 is above block2
to_right(block1,block2)	*Boolean domain:* true if block2 is to the right of block1
alignment(block1,block2)	*Nominal domain:* only_left_col, only_right_col, only_middle_col, both_columns, only_upper_row, only_lower_row, only_middle_row, both_rows

The following rules are used as background knowledge, in order to automatically associate information on page order to layout blocks.

at_page(X)=first □ part_of(Y,X), page(Y)=first
at_page(X)=intermediate□ part_of(Y,X), page(Y)=intermediate
at_page(X)=last_but_one □ part_of(Y,X), page(Y)=last_but_one
at_page(X)=last □ part_of(Y,X), page(Y)=last

Three long papers appeared in the January 1996 issue of the IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) have been considered. The papers contain thirty-seven pages, each of which has a variable number of layout components (about ten on average). Layout components can be associated with at most one of the

following eleven logical labels: *abstract*, *affiliation*, *author*, *biography*, *caption*, *figure*, *index_term*, *page_number*, *references*, *running_head*, *title.*

Learning rules for the recognition of semantically relevant layout components in a document raises issues concerning the induction of recursive theories. Simple and mutual concept dependencies are to be handled, since the logical components refer to a part of the document rather than to the whole document and may be related to each other. For instance, in case of papers published in journals, the following dependent clauses:

running_head(X) ← *top_left (X), text(X), even_page_number(X)*
running_head(X) ← *top_right (X), text(X), odd_page_number(X)*
paragraph(Y) ← *ontop(X,Y), running_head(X), text(Y)*

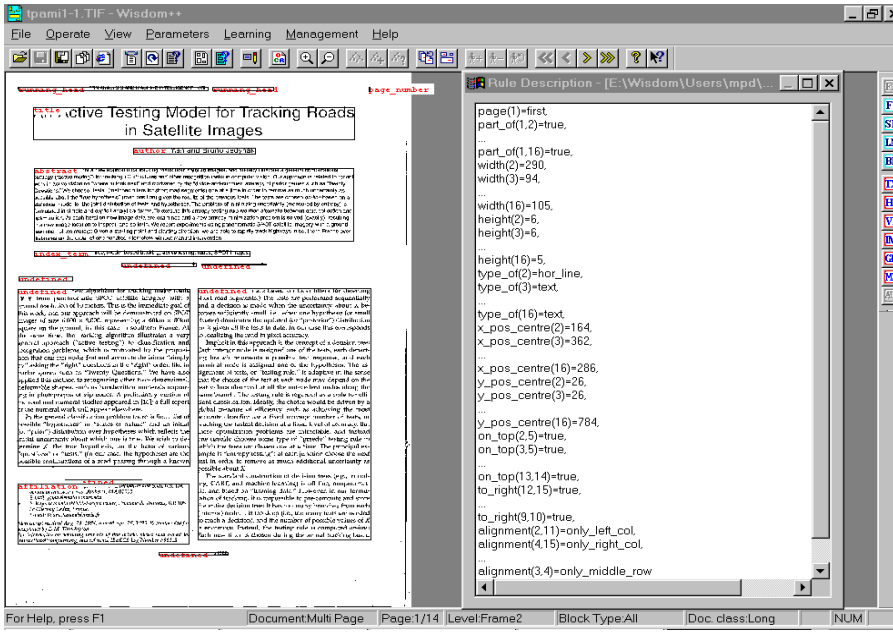


Fig. 3. Layout of the first page of a multi-page document (left) and its partial description in a first-order logic language (right).

express the fact that a textual layout component at the top left (right) hand corner of an even (odd) page is a running head, while a textual layout component below a running-head is a paragraph of the paper. Moreover, the recursive clause

paragraph(Y) ← *ontop(X,Y), paragraph(X), text(Y)*

is useful to classify all textual layout components below the upper-most paragraph. Therefore, document understanding seems to be the kind of application that may benefit of learning strategies for multiple predicate learning. Generally, the main benefits are:

! *Learnability of correct concept definitions*. For instance, some relational systems that do not take concept dependencies into account such as FOIL [28], cannot

learn the definitions of "appending two lists" and "reversing a list" independently, since the former concept is essential to give a reasonably compact definition of the second concept.

! *Rendering explicit some concept dependencies*, which would be otherwise hidden in a set of flat, independent rules. A correct logical theory structured around a number of dependent concepts does not contain those redundancies of its equivalent theory with independent concepts, therefore it is more comprehensible and easier to be validated by experts.

By running ATRE on the training set described above, the following theory is returned:

1. *logic_type(X)= running_head □ y_pos_centre(X)□[18 .. 39], width(X)□[77 .. 544]*
2. *logic_type(X)= page_number □ width(X) □ [2 .. 8], y_pos_centre(X)□[19 .. 40]*
3. *logic_type(X)= figure □ type_of(X)=image, at_page(X)=intermediate*
4. *logic_type(X)= figure □ type_of(X)=graphic*
5. *logic_type(X)= abstract □ at_page(X)=first, width(X)□[487 .. 488]*
6. *logic_type(X)= affiliation □ at_page(X)=first, y_pos_centre(X)□[720 .. 745]*
7. *logic_type(X)= caption □ height(X)□[9 .. 75], alignment(Y,X)=only_middle_col, logic_type(Y)= figure, type_of(X)=text*
8. *logic_type(X)= author □ at_page(X)=first, y_pos_centre(X)□[128 .. 158]*
9. *logic_type(X) = references □ height(X)□[332 .. 355], x_pos_centre(X)□[153 .. 435]*
10. *logic_type(X) = title □ at_page(X)=first, height(X)□[18 .. 53]*
11. *logic_type(X)=biography □ at_page(X)=last, height(X)□[65 .. 234]*
12. *logic_type(X)=caption □ height(X)□[9 .. 75], on_top(Y,X) , logic_type(Y)=figure, type_of(X)=text, to_right(Z,Y)*
13. *logic_type(X)=index_term □ height(X)□[8 .. 8], y_pos_centre(X)□[263 .. 295]*
14. *logic_type(X)=caption □ alignment(X,Y)=only_lower_row, height(X)□[9 .. 9]*
15. *logic_type(X)=caption □ on_top(Y,X), logic_type(Y)=figure, type_of(X)=text, alignment(Y,Z)=only_right_col*
16. *logic_type(X) = caption □ height(X)□[9 .. 75], on_top(X,Y), logic_type(Y)=figure, type_of(X)=text, type_of(Y)=graphic*
17. *logic_type(X)=caption □ height(X)□[9 .. 75], alignment(Y,X)=only_left_col, alignment(Z,Y)=only_left_col, logic_type(Z)=caption, width(Z)□[467 .. 546]*

Clauses are reported in the order in which they are learned. They have been filtered out from candidate clauses during the evaluation step performed according to a composite preference criterion that minimizes the number of negative examples covered, maximizes the number of positive examples covered, and minimizes the cost of the clause. The theory contains some concept dependencies (see clauses 7 and 12) as well as some kind of recursion (se clause 17). Learned concepts belong to two distinct layers: *abstract, affiliation, author, biography, figure, index_term, page_number, references, running_head,* and *title* are in one layer, while *caption* in the other. During the learning process, it was not necessary to recover theory consistency, and from our experience, the cases in which the recovery is required are rare. Surprisingly, some expected concept dependencies were not discovered by the system, such as that relating the running head to the page number:

logic_type(X)= page_number □ to_right(X,Y), logic_type(Y)= running_head
logic_type(X)= page_number □ to_right(Y,X), logic_type(Y)= running_head

The reason is due to the semantics of the descriptor *to_right*, which is generated by WISDOM++ only when two layout components are at a maximum distance of 100 points, which is not the case of articles published on the PAMI transactions. Same consideration applies to other possible concept dependencies (e.g., title-authors-abstract).

In order to test the predictive accuracy of the learned theory, we considered the fourth long article published in the same issue of the transactions used for training. WISDOM++ segmented the fourteen pages of the article into 169 layout components, sixteen of which (i.e., less than 10%) could not be properly labeled using by the learned theory (omission errors). No commission error was observed. This is important in this application domain, since commission errors can lead to totally erroneous storing of information. Finally, it is important to observe that many omission errors are due to near misses. For instance, the running head of the first page is not recognized simply because its centroid is located at point 40 along the vertical axis, while the range of *y_pos_center* values determined by ATRE in the training phase is [18..39] (see clause 1). Significant recovery of omission errors can be obtained by relaxing the definition of flexible matching between definite clauses [12].

7 Conclusions and Future Work

In this paper we have discussed and proposed computational solutions to some relevant issues raised by the induction of recursive theories in the normal ILP setting. A separate-and-parallel-conquer search strategy has been adopted to synchronize and interleave the learning of clauses supplying predicates with mutually recursive definitions. A novel generality order, called generalized implication, has been imposed to the search space of clauses in order to cope with recursion in a more suitable way. A layering technique based on the collapsed dependency graph has been investigated to recover the consistency of a partially learned theory. These ideas have been implemented in the ILP system ATRE and tested in the specific context of the document understanding problem. Experimental results obtained on a set of real-world multi-page documents empirically prove the validity of our approach to multiple predicate learning as well as the importance of taking predicate dependencies into account in the chosen domain application. In the future, we mean to refine the solutions being proposed in order to remove some inefficiency. In particular, it is necessary to optimize the separate-and-parallel-conquer search strategy with the aim of preventing it from exploring the specialization hierarchies repeatedly during the learning process. Moreover, we plan to investigate further the reasons causing the unsuccessful discovery of some expected concept dependencies. Finally, it is worth to test the system performance in case of document understanding problems with a richer set of logical components to be recognized.

References

1. Aha, D.W., Lapointe, S., Ling, C.X., Matwin, S.: Learning recursive relations with randomly selected small training sets. Proc. 11th Int. Conf. on Machine Learning, (1994) 12-18
2. Apt, K.R.: Logic programming. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, Vol. B. Elsevier, Amsterdam (1990) 493-574
3. Boström, H.: Specialization of recursive predicates. In Lavrac, N., Wrobel, S. (eds.): Machine Learning ECML-95. Lecture Notes in Artificial Intelligence, Vol. 912. Springer-Verlag, Berlin (1995) 92-106
4. Buntine, W.: Generalised subsumption and its applications to induction and redundancy. Artificial Intelligence, Vol. 36 (1988) 149-176
5. Cameron-Jones, R.M., Quinlan, J.R.: Avoiding pitfalls when learning recursive theories. Proc. 12th Int. Joint Conf. on Artificial Intelligence, (1993) 1050-1055
6. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Transactions on Knowledge and Data Engineering 1(1) (1989) 146-166
7. Cohen, W. W.: Learnability of restricted logic programs. In: Muggleton, S. (ed.): Proc. 3rd Int. Workshop on Inductive Logic Programming, (1993) 41-71
8. De Raedt, L.: Interactive Theory Revision. Academic Press, London (1992)
9. De Raedt, L., Lavrac, N., Dzeroski, S.: Multiple predicate learning. Proc. 13th Int. Joint Conf. on Artificial Intelligence, (1993) 1037-1042
10. De Raedt, L., Lavrac, N.: The many faces of inductive logic programming. In: Komorowski, J., Ras, Z.W. (eds.): Methodologies for Intelligent Systems. Lecture Notes in Artificial Intelligence, Vol. 689. Springer-Verlag, Berlin (1993) 435-449
11. De Raedt, L., Dehaspe, L.: Clausal discovery. Machine Learning 26(2/3), (1997) 99-146
12. Esposito, F., Caggese, S., Malerba, D., Semeraro, G.: Classification in noisy domains by flexible matching. Proc. European Symposium on Intelligent Techniques, (1997) 45-49
13. Esposito, F., Malerba, D., Lisi, F.A.: Machine Learning for Intelligent Processing of Printed Documents. In: Ras, Z.W., Skowron, A. (eds.): Journal of Intelligent Information Systems 16. Kluwer Academic Publishers (2000) 175-198
14. Flach, P.: A framework for inductive logic programming. In: Muggleton, S. (ed.): Inductive Logic Programming, Vol. 38 of Apic Series. Academic Press, London (1992) 193-211
15. Flener, P., Yilmaz, S.: Inductive Synthesis of Recursive Logic Programs: Achievements and Prospects. Journal of Logic Programming 41(2/3), Special Issue on Synthesis, Transformation and Analysis, (1999) 141-195
16. Giordana, A., Saitta, L., Baroglio, C.: Learning simple recursive theories. In: Komorowski, J., Ras, Z.W. (eds.): Methodologies for Intelligent Systems. Lecture Notes in Artificial Intelligence, Vol. 689. Springer-Verlag, Berlin (1993) 425-434
17. Idestam-Almquist, P.: Efficient induction of recursive definitions by structural analysis of saturations. In: De Raedt, L. (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 192-205
18. Jorge, A., Brazdil, P.: Architecture for iterative learning of recursive definitions. In: De Raedt, L. (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 206-218
19. van Leeuwen, J.: Graph Algorithms. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, Vol. A. Elsevier, Amsterdam (1990) 525-631
20. Malerba, D., Semeraro, G., Esposito, F.: A multistrategy approach to learning multiple dependent concepts. In: Nakhaeizadeh, G., Taylor, C. (eds.): Machine Learning and Statistics: The interface. John Wiley & Sons, New York (1997) 87-106
21. Malerba, D., Esposito, F., Lisi, F.A.: Learning Recursive Theories with ATRE. In: Prade, H. (ed.), Proc. 13th Europ. Conf. on Artificial Intelligence. John Wiley & Sons, Chichester (1998) 435-439
22. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997)

23. Mofizur, C.R., Numao, M.: Top-down induction of recursive programs from small number of sparse examples. In: De Raedt, L. (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 236-253
24. Nagy, G., Seth, S.C., Stoddard, S.D.: A prototype document image analysis system for technical journals. IEEE Computer 25(7), (1992) 10-22
25. Nienhuys-Cheng, S.-W., de Wolf, R.: A complete method for program specialization based upon unfolding. Proc. 12th Europ. Conf. on Artificial Intelligence (1996) 438-442
26. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 5. Edinburgh University Press, Edinburgh (1970) 153-163
27. Plotkin, G.D.: A further note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 6. Edinburgh University Press, Edinburgh (1971) 101-124
28. Quinlan, J.R.: Learning Logical Definitions from Relations. Machine Learning 5 (1990) 239-266
29. Rouveirol, C.: Flattening and saturation: Two representation changes for generalization. Machine Learning 14(2) (1994) 219-232

Extending K-Means Clustering to First-Order Representations

Mathias Kirsten[1] and Stefan Wrobel[2]

[1] German National Research Center for Information Technology, GMD - AiS.KD,
Schloß Birlinghoven, D-53754 Sankt Augustin,
mathias.kirsten@gmd.de
[2] Otto-von-Guericke-Universität Magdeburg, IWS,
P.O.Box 4120, D-39106 Magdeburg,
wrobel@iws.cs.uni-magdeburg.de

Abstract. In this paper, we present an in-depth evaluation of two approaches of extending k-means clustering to work on first-order representations. The first-approach, k-medoids, selects its cluster center from the given set of instances, and is thus limited in its choice of centers. The second approach, k-prototypes, uses a heuristic prototype construction algorithm that is capable of generating new centers. The two approaches are empirically evaluated on a standard benchmark problem with respect to clustering quality and convergence. Results show that in this case indeed the k-medoids approach is a viable and fast alternative to existing agglomerative or top-down clustering approaches even for a small-scale dataset, while k-prototypes exhibited a number of deficiencies.

1 Introduction

K-means clustering [16] has been a very popular technique for partitioning sets of objects and still is an interesting subject for research as the number of publications indicate [7,11,8]. In recent years the original restriction to numerical data has been relaxed by introduction of an extended k-means algorithm by [11] which is able to handle symbolic data as well. Nevertheless the family of k-means methods so far has been limited to propositional data only. This may turn out as a great disadvantage in domains where a larger expressive power is needed for adequate representation of the data.

In this paper, we therefore study two approaches that extend k-means clustering to work on first-order representations. The first approach, k-medoids, selects its cluster center from the given set of instances, and is therefore limited in its choice of possible centers. The second approach, k-prototypes, uses a heuristic prototype construction algorithm that is capable of generating new centers. The two approaches are empirically evaluated on a standard benchmark problem with respect to clustering quality and convergence.

The paper is organized as follows. In section 2, we outline the k-means optimization task for numerical and symbolic propositional data. In section 3, we present two approaches to extend k-means to work on first-order representations,

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 112–129, 2000.

namely k-medoids and k-prototypes, including a heuristic prototype construction algorithm. Results from our empirical evaluation are reported in section 4. In the related work section, we discuss the relation to other first-order clustering systems. We conclude with a summary and some pointers to future work. The appendix gives a detailed description of the prototype construction algorithm.

2 The *K*-Means Algorithm

The general algorithm was introduced by [4] ([16] and [1] first named it k-means) and has become widely popular since. It is defined as follows: given a set of n instances $\mathcal{I}$ from an instance space X, a natural number $1 \leq k \leq n$, and a distance measure $d(I_1, I_2) \rightarrow I\!R^{\geq 0}, I_1, I_2 \in X$, find a clustering $\mathcal{C} = \{C_1, \ldots, C_k\}$ into k non-empty disjunct clusters $C_l, l \in \{1, \ldots, k\}$, with $C_l \cap C_j = \emptyset$ and $\bigcup_l C_l = \mathcal{I}$ such that the overall sum of squared distances between instances and their clusters' center Q_l is minimized. Following the lines of [11], we can use indicator variables $w_{i,l}$ which take the value 1 iff instance I_i is in C_l, and 0 otherwise. Then the optimization task can be written as:

$$\text{Minimize} \qquad P(W, Q) = \sum_{l=1}^{k} \sum_{i=1}^{n} w_{i,l} d(X_i, Q_l) \tag{1}$$

$$\text{with } (W)_{n,k} = \begin{pmatrix} w_{1,1} & \cdots & w_{1,k} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \cdots & w_{n,k} \end{pmatrix}, \quad \sum_{l=1}^{k} w_{i,l} = 1, 1 \leq i \leq n, \text{ and } w_{i,l} \in \{0, 1\}.$$

To achieve the minimization, k-means iterates through a three-step process until the $P(W, Q)$ converges to some (local) minimum:

1. Select or construct a set of k instances $Q^{(0)} = \{Q_i^{(0)} | Q_i^{(0)} \in D, 1 \leq i \leq k, \forall i \neq j \; Q_i^{(0)} \neq Q_j^{(0)}\}$ (the cluster centers), and set $t = 0$.
2. Keep $Q^{(t)}$ fixed and compute the $W^{(t)}$ that minimizes $P(W, Q^{(t)})$ – i.e. regarding $Q^{(t)}$ as the cluster centers, assign each instance to the cluster of its nearest cluster center.
3. Keep $W^{(t)}$ fixed and generate $Q^{(t+1)}$ such that $P(W^{(t)}, Q^{(t+1)})$ is minimized – i.e. construct new cluster centers according to the current distribution of instances $\mathcal{I}$.
4. In case of convergence or if a given stopping criterion is fulfilled, print the result and terminate, otherwise set $t = t + 1$ and go to step 2.

As can be seen from equation (1) and step 3 in the above process, the two crucial operations are the distance computation between elements of the instance space X, and the computation of the new cluster centers $Q^{(t+1)}$ from a given distribution of the instances $\mathcal{I}$. Both operations depend strongly on the underlying representation formalism as well as on the elementary value types allowed within.

For numerical domains represented in attribute-value logic the (weighted) Euclidean distance is commonly chosen as the natural distance measure. With this distance measure, computation of the mean of a cluster's instances returns the cluster's center, fulfilling the minimization condition of step 3.

[11] (and also [8]) showed that on propositional domains the k-means method can be extended to symbolic data by employing a simple matching distance measure together with a majority-vote strategy to compute the cluster "center" (i.e. the Q that minimizes $P(W, Q)$). In more detail, the distance between two instances $I_1, I_2 \in X$, with $I_1 = (a_{1,1}, \ldots, a_{1,m})$ and $I_2 = (a_{2,1}, \ldots, a_{2,m})$ consisting of symbolic arguments only, can be defined as:

$$d(I_1, I_2) = \sum_{i=1}^{m} d(a_{1,i}, a_{2,i}), \text{ with } d(a, b) = \begin{cases} 0, & \text{if } a = b \\ 1, & \text{otherwise} \end{cases}$$

With this distance measure, a majority vote strategy that selects an argument's most frequent value can be used to compute the cluster "center". So, given a cluster (respectively a set of instances) $\{I_1, \ldots, I_o\}$, with $I_l = (a_{l,1}, \ldots, a_{l,m}), 1 \leq l \leq o$, its center $Q = (q_1, \ldots, q_m)$ is computed by assigning $q_j, 1 \leq j \leq m$ the value most frequently encountered in $\{a_{1,j}, \ldots, a_{o,j}\}$.

This meets the minimization condition, as has been shown in [11]. Furthermore, by combining both the methods for numerical and categorical data, k-means can be extended to handle domains with mixed value types. To distinguish the methods for different argument types, [11] uses "k-means" for strictly numerical data, "k-modes" for categorical values, and "k-prototypes" for the mixture of both.

Nevertheless, k-means methods have been limited to propositional domains. In the following we will hence elaborate the components necessary to extend the existing approaches to first-order domains.

3 *K*-Means for First-Order Representations

The task of k-means clustering on a first-order representation as used in this paper can be defined as follows:

Given:

- a set of *instances* $\mathcal{I} \subseteq X$
- a *distance function* $d : X \times X \rightarrow \mathbb{R}$
- a *center function* $\mathsf{center} : 2^X \rightarrow X$
- background knowledge $\mathcal{B}$
- the number of clusters k
- an initial set of k *seeds* $Q^0 \subseteq X$

Find:

- a set of clusters $\mathcal{C} = \{C_1, \ldots, C_k\} \subseteq 2^X$
- that maximizes given *quality criteria*

While the set of seeds Q^0 can be initialized by randomly drawing k instances from $\mathcal{I}$, which is the defacto standard according to [7], the crucial issues for getting k-means to work with first-order representations are the choice of an appropriate distance measure and the definition of the function finding a clusters' center.

Distance Measures For propositional numerical data, the (weighted) Euclidean distance is a natural choice, as are simple matching similarities for categorical data. As a matter of fact, similarly natural choices for multirelational data are not at hand. Nevertheless, several distance measures for first-order representations have been proposed [18,12,6,22,21,9]. Because the distance measure employed in RIBL [9] has exhibited excellent results on several domains [6,10], we decided to use it as the basis for this work.

Computing Cluster Centers As reported in section 2, for propositional data solutions exist that compute the exact cluster center (i.e. the point that minimizes the sum of squared distances $\sum_{I_l \in C_i} d(I_l, Q_i)$). However, until now, no such approach is available for multirelational representations. Furthermore, the phrase "cluster center" allows no geometric or intuitive notion in case of multirelational data. Nevertheless, one can think of several options what the center of a set of multirelational instances could be:

One option is the LGG of the set of instances [19]. Unfortunately, for our purposes, the LGG exhibits three shortcomings. Firstly, computing the LGG of a set of instances and their background knowledge is computationally very expensive, and the LGG can grow exponentially with the number of instances. Secondly, the distance measure of RIBL is only defined between instances comprising sets of ground facts, i.e. no variables are allowed. And thirdly, it is very likely that LGG's would be very general (and often just the most general clause), so two different sets of instances could yield the same LGG, thus spoiling the further k-means process (e.g. assigning cluster-memberships).

Alternatively, the union of all instances in the set (respectively their ground facts) could do as the center. But in fact, such a "monster-instance" would hardly be suitable for further calculations, and the idea to achieve better computational complexity by using a cluster center instead of a cluster's set of instances would be spoiled totally.

Third, the most central instance of the set, its medoid, could be used as an approximation for the center. The most central instance is the instance whose sum of squared distances to all other instances in the set is minimal. This approach is already known as the k-medoids method [13,14]. Depending on the domain, the results may suffer from the restrictions on possible cluster centers. If the (locally) optimal solution for a cluster center does not coincide with existing instances, the optimal solution cannot be found. Furthermore, the convergence process may fail when an oscillating state between either sides of an optimum is reached.

Fourth, an averaged instance, constructed by copying the structure from the instances of the set (or by overlapping several structures, if more than one structure exists). The facts in the structure can be computed recursively, by building averaged objects or values for each argument of a fact. On the downside of this approach, it is not yet clear, if such a constructive approach can satisfy the criterion for the cluster center, or if a good approximation can be constructed anyhow.

Having detailed the different options, it becomes clear that only the latter two are computationally reasonable alternatives.

3.1 *K*-Medoids

Similar to the k-means methods, the k-medoids algorithm [14] follows the process detailed in section 2. It differs in the computation of the cluster centers Q^{t+1} as the medoid approach restricts Q to be a subset of the existing instances $\mathcal{I}$ whereas k-means allows Q to be a subset of the whole instance space X. So, k-medoid approximates the actual cluster center by taking an existing instance that minimizes the given center criterion (e.g. the sum of squared distances) within its cluster.

Obviously, the whole process relies on distance information only, and in contrast to k-means, is hence applicable to any representation for which an appropriate distance measure is available. On the downside, when making no prior assumptions about the representation used, the computational complexity suffers from the search for the best approximations which is proportional to the squared number of instances in a cluster.

3.2 *K*-Prototypes

The center finding function for the k-prototypes method should be able to find better approximations to the actual cluster center, as it is not limited in the choice of possible centers. In fact, an ideal center function constructs prototypes that minimize the center criterion (e.g. sum of squared distances). For non-propositional representations with structured and complex objects this is hard to achieve, and depends strongly on the underlying distance measure.

To construct a prototype for a set of multirelationally represented instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ our approach uses a recursive descent strategy in order to take into account not only the arguments of the instances but also the available background knowledge. Each instance I_j in our representation consists of one target fact F_j and the set of related facts from the background knowledge $\mathcal{B}$.

For a start we consider the set of target facts $F = \{F_1, \ldots, F_n\}$ with a given predicate symbol and arity q. The prototype is then computed by generating a new q-ary atom and iteratively calculating the values of its arguments. In case an argument is of an atomic type like number or constant, this can be done by falling back to the known prototype functions for propositional representations. Otherwise, if an argument is a link to another object (respectively fact), all

objects referred to by the argument are gathered and a new (sub-)prototype is built from these objects recursively.

Following this recursively descending process, the prototype's structure should eventually resemble the structure of the original instances. As the structure may contain cycles, a depth bound is employed to avoid infinite recursion.

Figure 1 shows an examplary prototype for a set of three instances I_1, I_2, I_3 (see appendix A for a detailed description).

Instances:

I_1 := { package(*set*1,125,*personal*),
cheese(*set*1,*camembert*,150,*france*),
wine(*set*1,*mouton*,1988,0.75),
wine(*set*1,*gallo*,1995,0.5),
vineyard(*gallo*,*famous*,*large*,*use*),
vineyard(*mouton*,*famous*,*small*,*france*)}

I_2 := { package(*set*25,195,*mail*),
cheese(*set*25,*roque fort*,200,*france*),
cheese(*set*25,*ricotta*,100, *italy*),
wine(*set*25,*mouton*,1995,0.75),
vineyard(*mouton*,*famous*,*small*,*france*)}

I_3 := { package(*set*10,150,*personal*),
cheese(*set*10,*camembert*,300,*france*)}

Resulting prototype:

Pr := { package(*new_id*1,156,*personal*),
cheese(*new_id*1,*camembert*,187,*france*),
wine(*new_id*1,*new_id*2,1992,0.6667),
vineyard(*new_id*2,*famous*,*small*,*use*)}

Fig. 1. Example of generated prototype from a set of three instances.

In order to apply the prototype algorithm to data sets containing lists, one needs to find the list with the minimal sum of distances between itself and all other lists. This leads to two alternatives. First, to use the medoid from the set of lists as an approximation. Second, to construct a new list that comes close to the desired minimization property.

Unfortunately, the medoid approach is computationally too expensive as soon as a more sophisticated distance measure like edit distance is employed, so this option is not further considered here.

The constructive approach we employ is less costly but does not take the neighborhoods of arguments into account as it is done by the edit distance. The length of the prototype list is set to the averaged length of the set of lists. Then, starting at the first position, each element of the prototype list is determined by a majority vote over the elements of the lists at the specific position. This can be done in $O(nm)$ time, where m is the average number of elements in a list and n is the number of lists.

4 Empirical Evaluation

In the preceding sections, we have presented two different adaptations of the k-means clustering method to first-order data, the k-medoid and the k-prototype algorithms. Similar to the original k-means, as pointed out above, both of these have to be regarded as optimization methods that search for a clustering with maximal quality with respect to the chosen distance measure. For k-means, it is known that excellent cluster quality results with very fast convergence in most domains. Consequently, the primary goal of the experiments reported subsequently was to find out whether these positive properties of propositional k-means as an optimization method carry over to first-order k-medoid and k-prototype.

In particular, we have examined the following questions.

- Is clustering quality on a par with the results of RDBC [15], an agglomerative clustering method based on the same distance measure[1]? Does prototype construction offer improved results by using better centers than those available in the data?
- Do k-medoid and/or k-prototype converge fast, and can they thus improve on agglomerative clustering where the entire hierarchy has to be built? Are the convergence properties of k-prototypes better than k-medoid which might oscillate due to lack of proper centers in the data?

4.1 Experimental Setup

In order to allow detailed evaluation and easy comparison with other researchers' results, we have selected the standard "mutagenicity" benchmark dataset [23] for our experiments. In this application, the goal is to classify chemical compounds into "non-mutagenic" and "mutagenic". For the purposes of clustering, we have removed this class information and just worked with the compound descriptions, consisting of the usual set of twenty extensionally defined predicates encoding structural and non-structural information about the compounds. For most experiments reported here, the full background knowledge "BG4" was used; however, in order to be able to separately test the effect of list construction on the performance of k-prototypes, the list-free "BG3" background knowledge was used in one experiment also. Except where explicitly marked, we have worked with the larger ("regression-friendly") dataset consisting of 188 compounds.

All results reported below were obtained by averaging, for each value of k from 2 to 15, 10 different runs with independently chosen random starting configurations (k-medoid), respectively 5 different runs (k-prototypes). If error bars are shown in a figure, they are standard deviations across the different runs. To examine convergence, we let each method run continue for 50 iterations, in which period all runs had reached convergence or cycled around a local optimum.

[1] Note that in past work [15], we have already compared RDBC's agglomerative clustering to other non-distance-based clustering approaches, with favorable results.

The reported quality measure values in summary evaluations are the arithmetic mean of the last 10 iterations (40 to 50).

In addition, the same data was applied to several versions of RDBC [15] of which we report the results for the average link and the single link runs. The RDBC algorithms normally construct cluster hierarchies and turn them into flat partitions via a threshold algorithm that automatically tries to find the optimal number of clusters. For sake of comparison, we forced the threshold algorithm to produce a predefined number of clusters from a clustering hierarchy and calculated the quality measures accordingly. Since RDBC is not iterative and does not depend on starting conditions all figures reported refer to this single run of the algorithm and its resulting single clustering.

4.2 Quality Measures

Since k-means type algorithms optimize squared error with respect to their cluster center instead of average intra-cluster distance (which is used by the average-link variant of RDBC), both measures were considered and computed as follows[2] ($\mathsf{center}(C_i)$ denotes the selected medoid for k-medoid, respectively the computed prototype for k-prototypes):

$$\mathsf{intra\_sim}(C) = \frac{\sum_{i=1}^{k} \sum_{I_j, I_k \in C_i, j \neq k} \mathsf{similarity}(I_j, I_k)}{\sum_{i=1}^{k} |C_i|(|C_i|-1)}$$

$$\mathsf{avg\_squared\_dist}(C) = \frac{\sum_{i=1}^{k} \sum_{I_j \in C_i} d(I_j, \mathsf{center}(C_i))^2}{\sum_{i=1}^{k} |C_i|}$$

In addition, since the mutagenicity domain really is a classification problem, it is possible to "abuse" the clustering (built without class information) for classification by labeling (a posteriori) each cluster with its majority class, and classifying a test instance according to the label of the closest cluster. These figures thus can be compared to those found in literature for classificatory ILP systems [23,24].

4.3 Results on Cluster Quality

As shown in Figure 2, experimental results[3] confirmed our expectations regarding the general quality of k-medoid clustering results to be on a par with the quality of agglomerative clustering . However, for k-prototypes, Figure 2 shows surprising results. While for small k, it performs comparably to k-medoid clustering, its performance drops for larger k. Given that a larger number of clusters each with a smaller number of elements should allow even more exact cluster prototype construction and smaller distances, this result is surprising.

[2] Note that intra-cluster similarity as reported here normalizes the sum of similarities by the number of similarities, not by cluster size. The latter did not show any qualitative change in the results.

[3] Results shown here are for average intra-cluster distance, but similar results were obtained for square error.

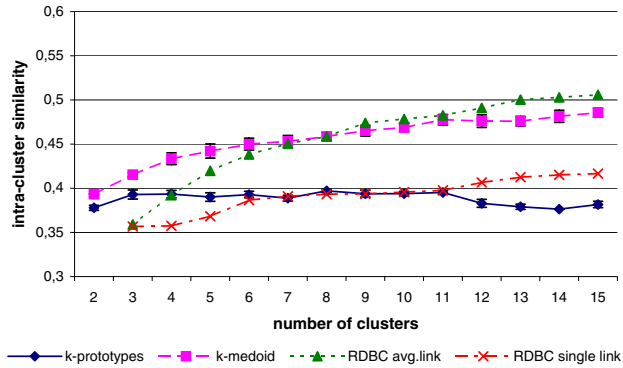


Fig. 2. Intra-cluster similarity for $k \in \{2, \ldots, 15\}$ clusters.

Upon inspection of the runs, it could be seen that part of the reason for this outcome is the "loss" of clusters that happens during the process. This loss occurs in step 3 of the clustering process (see section 2) where each instance is assigned to the cluster of the prototype (or medoid) nearest to it. Some of the prototypes end up with no instances assigned to them, thus forming empty clusters which have to be removed to proceed with the process. As it turns out, only 2 to 4 effective clusters remained after 50 iterations. When comparing k-prototypes to k-medoid on effective number of clusters (Figure 3), the counter-intuitive drop in performance disappears, but overall performance is still lower, indicating that indeed the proposed method of prototype computation does not accurately construct a cluster center.

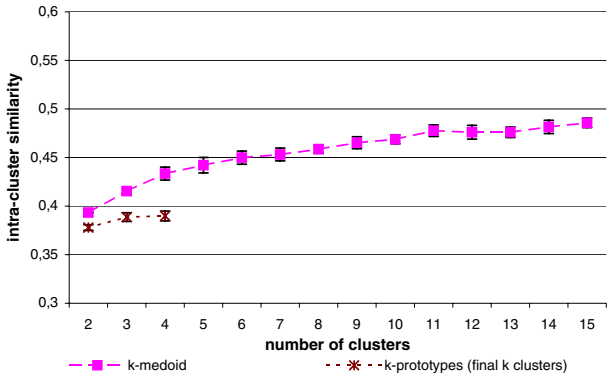


Fig. 3. Intra-cluster similarities of k-prototypes and k-medoids in regard to remaining number of clusters.

Since k-medoids performed well on this dataset, apparently even in 188 instances appropriate cluster centers were available. We therefore conducted additional experiments with the 42-instance dataset to see if lack of appropriate centers would force down performance there, and indeed (not shown graphically), for k larger than 9, a drop in k-medoid performance was observed. To complete the favorable results of k-medoid on this dataset, Figure 4 shows accuracy results compared to PROGOL [17], FOIL [20], and RIBL, where good accuracies are obtained even with a small number of clusters [4].

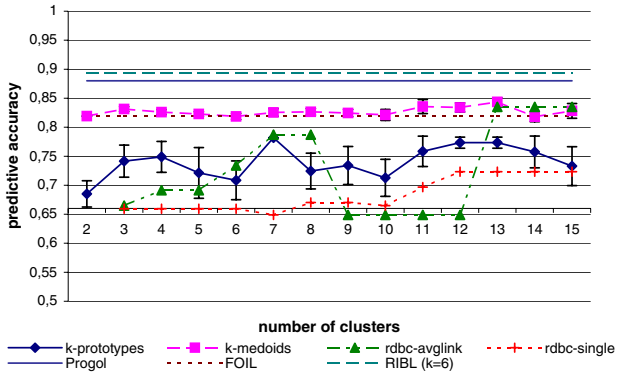


Fig. 4. Predictive accuracies on regression friendly data with BG4. Position of the X-axis indicating the accuracy of the default rule.

4.4 Results on Convergence

Ideally, an iterative clustering method should reach a stable solution after a few iterations, as is often the case for propositional k-means. This should be reflected in stable quality measures for upcoming iterations. To evaluate convergence, we therefore computed (summed across all k) the 40 standard deviations of the intra-cluster similarities within a window of 10 consecutive iterations, e.g. data point 1 shows the standard deviation of quality within iterations 1 to 10 (Figure 5). As can be seen, the convergence of the k-medoids algorithm happens within the first 20 iterations. K-prototypes, however, starts the convergence process with slightly smaller changes than k-medoids and settles down at a significantly higher level of deviation, i.e., could not reach convergence.

A closer look at the single runs reveals the reason, which lies in k-prototypes' strong tendency to go into cyclic states. Figure 6 shows an example of an 18-step

[4] The tree-based first-order clustering system TIC has also been applied to this domain [3], however, no experiments using the complete background knowledge BG4 have been reported.

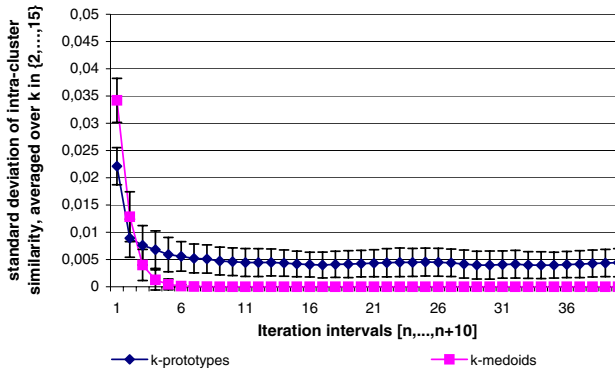


Fig. 5. Convergence properties of k-prototypes and k-medoids on regression friendly data with BG4, using the standard deviation of intra-cluster similarity on a window of 10 iterations, averaged over $k \in \{2, \ldots, 15\}$, as an indicator.

cycle reached after 9 iterations. As almost all runs end up in such cyclic states, strategies to resolve this dilemma are necessary.

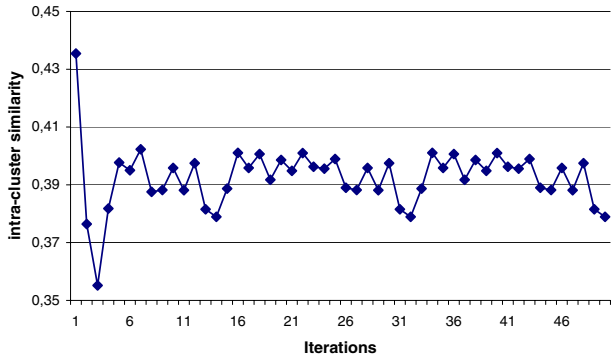


Fig. 6. Example of k-prototypes running into an 18-step cycle.

Evaluating the convergence of accuracy and squared-distance scores results in very similar figures, which are omitted here for sake of shortness.

4.5 Detailed Study of *K*-Prototype Properties

Since the above results indicate that k-prototypes does not appropriately construct cluster centers, we decided to further investigate why this is the case. To

investigate which parts of the construction are responsible for this deficiency, we singled out the list-construction part which uses an extremely simplistic approach that quite obviously does not mirror the edit-distance criterion used in the distance measure. However, as shown in Figure 7, experiments with the list-free background knowledge "BG3" did not show any marked difference.

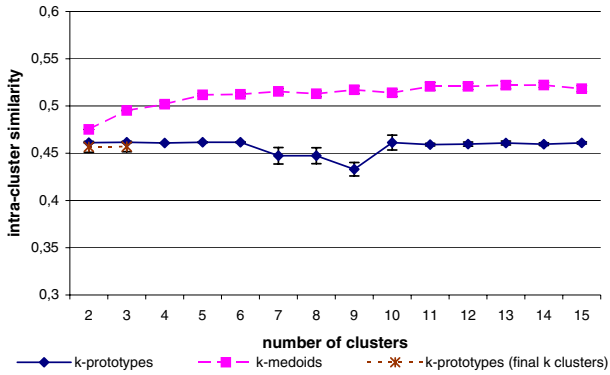


Fig. 7. Intra-cluster similarities of k-prototypes and k-medoids plotted in regard to the predefined number of clusters and in regard to the remaining number of clusters for k-prototypes.

To test whether this was due to non-center positioning of prototypes, we explicitly examined the average square distance to the cluster center. Figure 8 shows both intra- and inter-cluster distances of prototypes and medoids in comparison. Interestingly, k-prototypes *does* exhibit smaller intra-cluster distances than k-medoid. However, its inter-cluster distances are also lower than for the medoids approach, indicating that there exists a qualitative difference between the generated prototypes and actual instances — otherwise intra and inter-cluster distances would have to be more similar to those of the medoids.

5 Related Work

The presented k-prototypes and k-medoids approaches contrast with a number of previous first-order clustering systems, namely KBG [2], Cola-2 [5], TIC [3], and RDBC [15]. These systems cover different clustering strategies. RDBC and KBG follow a bottom-up clustering approach, and TIC builds its clustering trees in top-down manner, while the k-prototypes and k-medoid methods iteratively optimize an existing partitioning. Both the k-prototypes as well as the k-medoid method rely on a first-order distance measure and k-medoids compares favorably to the other methods in terms of cluster quality. Furthermore, in addition to the

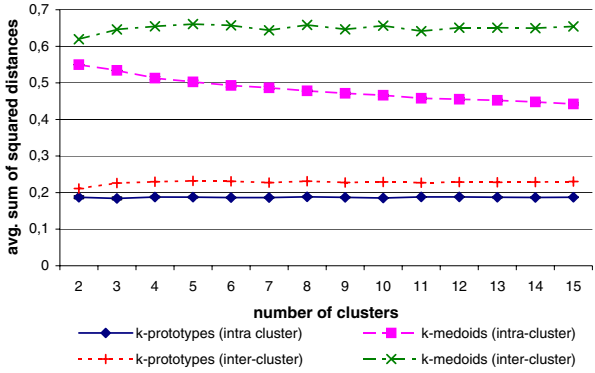


Fig. 8. Intra and inter-cluster distances of prototype and medoid algorithms for different numbers of clusters.

purely distance based approach taken in RDBC, k-prototypes and k-medoids return a prototype respectively the medoid of each cluster. Although less succinct than the cluster descriptions generated by conceptual clustering systems like KBG, COLA-2, or TIC, the set of prototypes/medoids can nevertheless be useful for subsequent (distance-based) processing.

In addition, this work is related to the extension of k-means to handle symbolic arguments as proposed in [11]. There, a majority vote strategy is used to build prototypes for propositional instances with symbolic arguments. The multirelational prototype construction within our k-prototypes algorithm adopts this approach to determine the symbolic arguments of facts in a prototype.

6 Conclusion and Future Work

In this paper, we have presented an in-depth evaluation of two approaches of extending k-means clustering to work on first-order representations. The first-approach, k-medoids, selects its cluster center from the given set of instances, and is thus limited in its choice of centers. The second approach, k-prototypes, uses a heuristic prototype construction algorithm that is capable of generating new centers.

The empirical evaluation of the two approaches on the mutagenicity domain presents a very clearcut conclusion. In this domain, both in terms of cluster quality and convergence, k-medoids is on a par with non-iterative state-of-the-art agglomerative clustering as previously examined in RDBC [15], while k-prototypes performed significantly worse and generated artefacts due to its way of constructing clusters. If these findings generalize to further domains, they show that indeed the k-medoids approach presented here is a viable alternative to existing agglomerative or top-down clustering approaches even in small-scale

datasets. To this end, more experiments should be done in the future with other domains.

As for prototype generation, improvements to the method presented here appear possible that might improve its center computation. If prototype generation did not simply use the most frequent fact, but instead selected the x most frequent facts to be part of the prototype, this might lead to better approximations of the cluster center (with respect to the goal of decreasing intra cluster and increasing inter cluster distance).

References

1. G. H. Ball and D. J. Hall. A clustering technique for summarizing multivariate data. *Behavioral Science*, 12:153–157, 1967.
2. G. Bisson. Conceptual Clustering in a First-Order Logic Representation. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 458–462. John Wiley, 1992.
3. H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In J. Shavlik, editor, *Proceedings of the Fiteenth International Conference on Machine Learning (ICML-98)*, pages 55–63. Morgan Kaufmann, 1998.
4. D. Cox. Note on grouping. *J. Am. Stat. Assoc.*, 52:543–547, 1957.
5. W. Emde. Inductive learning of characteristic concept descriptions from small sets to classified examples. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 103–121. Springer-Verlag, 1994.
6. W. Emde and D. Wettschereck. Relational Instance-Based Learning. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, 1996.
7. U. M. Fayyad, C. Rain, and P. S. Bradley. Initialization of iterative refinement clustering algorithms. In R. Agrawal, P. E. Stolorz, and G. Piatetsk-Shapiro, editors, *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 194–198. AAAI Press, 1998.
8. S. K. Gupta, K. Sambasiva Rao, and V. Bhatnagar. K-means Clustering Algorithm for Categorical Attributes. In M. K. Mohania and A. Min Tjoa, editors, *Proceedings of the First International Conference on Data Warehousing and Knowledge Discovery (DaWaK-99)*, volume 1676 of *Lecture Notes in Computer Science*, pages 203–208. Springer-Verlag, 1999.
9. T. Horváth, S. Wrobel, and U. Bohnebeck. Relational instance-based learning with lists and terms. Machine Learning (to appear).
10. T. Horváth, S. Wrobel, and U. Bohnebeck. Term comparisons in first-order similarity measures. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 65–79. Springer-Verlag, 1998.
11. Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Mining and Knowledge Discovery*, 2(3):283–304, 1998.
12. A. Hutchinson. Metrics on Terms and Clauses. In M. Someren and G. Widmer, editors, *Proceedings of the 9th European Conference on Machine Learning*, volume 1224 of *LNAI*, pages 138–145. Springer-Verlag, 1997.

13. L. Kaufmann and P. J. Rousseeuw. Clustering by means of medoids. In Y. Dodge, editor, *Statistical Data Analysis based on the L_1 Norm*, pages 405–416. Elsevier Science Publishers, 1987.
14. L. Kaufmann and P. J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley, 1990.
15. M. Kirsten and S. Wrobel. Relational distance-based clustering. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *LNAI*, pages 261–270. Springer-Verlag, 1998.
16. J. McQueen. Some methods of classification and analysis of multivariate observations. In L. K. Le Cam and J. Neyman, editors, *Proceedings of Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–293, 1967.
17. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
18. S.-H. Nienhuys-Cheng. Distance Between Herbrand Interpretations: A Measure for Approximations to a Target Concept. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *LNAI*, pages 213–226. Springer-Verlag, 1997.
19. G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
20. J. Quinlan and R. Cameron-Jones. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 3–20. Springer-Verlag, 1993.
21. J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 271–280. Springer-Verlag, 1998.
22. M. Sebag. Distance induction in first order logic. In N. Lavrač and S. Džeroski, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, LNAI, pages 264 – 272. Springer-Verlag, 1997.
23. A. Srinivasan, S. Muggleton, and R. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*, pages 199–230. Department of Computer Science, Katholieke Universiteit Leuven, 1995.
24. A. Srinivasan, S. Muggleton, M. Sternberg, and R. King. Theories for mutagenicity: a study in first-order and feature-based induction. *Artificial Intelligence*, 85:277 – 299, 1996.

A Heuristic Prototype Generation

To construct a prototype for a set of first-order instances $\mathcal{I} = \{I_1, \ldots, I_n\}$ our approach uses a recursive descent strategy in order to take into account not only the arguments of the instances' target fact but also the available background knowledge. Each instance I_j in our representation consists of one target fact F_j and the set of related facts from the background knowledge $\mathcal{B}$.

To illustrate the construction process, we give an example using a slightly extended version of the "present package" example introduced in [9]. The data comprises three instances of culinary packages together with the corresponding background knowledge. It is assumed that about each package, its price and

its delivery mode is known (personal, mail, or pick-up). So, each instance is represented by a (target) fact containing three arguments: identifier, price, and delivery mode plus the related background knowledge. As the contents of each package is known – it consists of different red wines and cheeses – and information about the different vineyards is given, this information makes up the background knowledge. The two packages are represented as follows:

$$F_1 := \text{package}(set1,125,personal)$$
$$F_2 := \text{package}(set25,195,mail)$$
$$F_3 := \text{package}(set10,150,personal)$$

In these target facts, the first argument is of type object and refers to further information in the background knowledge $\mathcal{B}$:

wine(*set*1, *mouton*, 1988, 0.75)
wine(*set*1, *gallo*, 1995, 0.5)
wine(*set*25, *mouton*, 1995, 0.75)

cheese(*set*1, *camembert*, 150, *france*)
cheese(*set*10, *camembert*, 300, *france*)
cheese(*set*25, *roque fort*, 200, *france*)
cheese(*set*25, *ricotta*, 100, *italy*)

vineyard(*gallo*, *famous*, *large*, *usa*)
vineyard(*mouton*, *famous*, *small*, *france*)

To compute the prototype of a set of instances, in our example $\{I_1, I_2, I_3\}$, a new but yet empty fact Pr is generated first.

$$Pr := \text{package}(arg_1,arg_2,arg_3)$$

The values for the arguments (arg_1, arg_2, and arg_3) are computed sequentially. In case an argument is of an atomic type like number or constant, its value can be computed by falling back to the known prototype functions for propositional representations.

In our example the second argument is of type integer and the third of symbolic type. Hence, arg_2 can be computed by calculating the mean of the argument's values and arg_3 is determined by majority vote on the third argument:

$$arg_2 = \left\lfloor \frac{125+195+150}{3} \right\rfloor = 156$$
$$arg_3 = \text{majority\_vote}(\{personal,mail,personal\}) = personal$$

If an argument is of type object, like the first argument of F_j in our example, two cases have to be distinguished.

In the first case the argument points *down* to objects one level deeper in the instances' structure and one has to decide (e.g. via a depth bound) wether (sub-)prototypes of the related objects should be generated or not. If (sub-)prototypes should be generated, a new id for linking to them must be provided. Otherwise, the argument's value can be determined by majority vote analogous to symbolic argument types.

In our example the first argument of F_j is of type object and allows the wine and cheese facts to refer to it. As we are still at the top-level, prototypes for

the underlying objects should be constructed. Hence, an ID for the new (sub-) protoypes is generated.

$$arg_1 = \mathsf{generate\_unique\_id}() = id_1$$

Having decided to build prototypes for the underlying objects, all facts the argument values refer to are gathered and sorted into subsets $(G_1, \ldots, G_r)$ according to their arity and predicate symbol. For each subset G_j a new prototype is constructed recursively.

Coming back to the example and gathering the background knowledge referring to I_1, I_2, and I_3, the related facts are grouped into subsets according to their predicate symbol and arity:

$$\begin{aligned} G_1 = \{&\text{cheese}(set1,\ camembert,\ 150,\ france),\\ &\text{cheese}(set10,\ camembert,\ 300,\ france),\\ &\text{cheese}(set25,\ roque\ fort,\ 200,\ france),\\ &\text{cheese}(set25,\ ricotta,\ 100,\ italy)\}\\ G_2 = \{&\text{wine}(set1,\ mouton,\ 1988,\ 0.75),\\ &\text{wine}(set1,\ gallo,\ 1995,\ 0.5),\\ &\text{wine}(set25,\ mouton,\ 1995,\ 0.75)\} \end{aligned}$$

For each subset a new (sub-)prototype is generated, in this case, one for wine and another one for cheese. All numeric and symbolic arguments are generated as described above, which results in the following prototypes:

$$\begin{aligned} Pr_{sub1} &= \text{cheese}(id_1, camembert,\ 187,\ france)\\ Pr_{sub2} &= \text{wine}(id_1,\ arg_{wine,2},\ 1992,\ 0.666667) \end{aligned}$$

In the second case, the argument points *up* to a fact one level above in the instances' structure. id_1 and id_2 from the (sub-)prototypes Pr_{sub1} and Pr_{sub2} are good examples. The value for the argument has been generated in the level above and allows the new prototypes to refer to their parent fact (Pr).

Following our example, we find that the second argument of Pr_{sub2} is of type object and links down to vineyard. Assuming that a depth bound is not yet reached and a new subprototype should be constructed, we generate another id and proceed as above by constructing a new prototype for the referred background knowledge (i.e. the vineyard facts):

$$\begin{aligned} arg_{wine,2} &= \mathsf{generate\_unique\_id}() &&= id_2\\ Pr_{sub2_{sub1}} &= \text{vineyard}(id_2,\ famous,\ small,\ usa) \end{aligned}$$

As the reader has possibly noticed, trying to compute the values for the last two arguments of $Pr_{sub2_{sub1}}$ using a simple majority vote results in a draw. Such a draw can be resolved by a random selection from the most frequent argument values or by using overall frequency in the dataset.

prototype (F,DepthBound,CurrentDepth,Link):
var
F : Set of q-tuples $\{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$
DepthBound : Depth bound for recursive prototype generation
CurrentDepth : Current depth level of the process
Link : Link to above level fact (possibly empty)

A : Vector of argument values $\{a_1, \ldots, a_n\}$

1. **for** $l := 1$ **to** q **do**
2. $a_i := \mathsf{f}_i[l], \forall i \in \{1, \ldots, n\}$ %% $\mathsf{f}_i[l]$: l-th argument of fact f_i
3. **case of** $type(a_i)$:
4. number: $\mathsf{arg}_l := \left(\sum_{i=1}^{n} a_i\right)/n$
5. constant: $\mathsf{arg}_l := \mathsf{majority\_vote}(A)$
6. list: $\mathsf{arg}_l := \mathsf{list\_prototype}(A)$
7. link: **if** (Link $\neq \{\}$) **then**
8. $\mathsf{arg}_l :=$ Link
9. **else if** (CurrentDepth $\geq$ DepthBound) **then**
10. $\mathsf{arg}_l := \mathsf{majority\_vote}(A)$
11. **else**
12. $\mathsf{arg}_l := \mathsf{generate\_unique\_id}()$
13. NewFacts := $\mathsf{gather\_related\_facts}(A, \mathsf{F})$
14. $(\mathsf{G}_1, \ldots, \mathsf{G}_r) := \mathsf{split\_into\_q\_atoms}(\mathsf{NewFacts})$
15. **for** $j := 1$ **to** r **do**
16. $\mathsf{prototype}(\mathsf{G}_j, \mathsf{DepthBound}, \mathsf{CurrentDepth} + 1, \mathsf{arg}_l)$
17. **end for**
18. **end if**
19. **end case**
20. **end for**
21. **return** $\mathsf{prototype\_instance}(\mathsf{arg}_1, \ldots, \mathsf{arg}_q)$

gather_related_facts (A,F):
var A : Vector of values $\{a_1, \ldots, a_n\}$
F : Set of q-atoms $\{\mathsf{f}_1, \ldots, \mathsf{f}_n\}$
1. **return** the set of all facts from the reduced background knowledge $B \setminus \mathsf{F}$ where a value from A appears in the position of an object-typed argument, i.e. return all facts referring to or referred by any of the values in A.

split_into_q_atoms (NewFacts):
var NewFacts : List of facts $\{\mathsf{nf}_1, \ldots, \mathsf{nf}_n\}$
1. **return** the list of subsets $(\mathsf{NewFacts}^1, \ldots, \mathsf{NewFacts}^r)$ that results from sorting and splitting the list of facts NewFacts according to their arity and predicate symbol. I.e. $\mathsf{NewFacts}^j$ contains only facts of the same arity and predicate symbol.

Fig. 9. Prototype algorithm for set of facts.

Theory Completion Using Inverse Entailment

Stephen H. Muggleton and Christopher H. Bryant

Department of Computer Science,
University of York,
York, YO1 5DD,
United Kingdom.

Abstract. The main real-world applications of Inductive Logic Programming (ILP) to date involve the "Observation Predicate Learning" (OPL) assumption, in which both the examples and hypotheses define the same predicate. However, in both scientific discovery and language learning potential applications exist in which OPL does not hold. OPL is ingrained within the theory and performance testing of Machine Learning. A general ILP technique called "Theory Completion using Inverse Entailment" (TCIE) is introduced which is applicable to non-OPL applications. TCIE is based on inverse entailment and is closely allied to abductive inference. The implementation of TCIE within Progol5.0 is described. The implementation uses contra-positives in a similar way to Stickel's Prolog Technology Theorem Prover. Progol5.0 is tested on two different data-sets. The first dataset involves a grammar which translates numbers to their representation in English. The second dataset involves hypothesising the function of unknown genes within a network of metabolic pathways. On both datasets near complete recovery of performance is achieved after relearning when randomly chosen portions of background knowledge are removed. Progol5.0's running times for experiments in this paper were typically under 6 seconds on a standard laptop PC.

1 Introduction

Suppose that an ILP system is being used to augment an incomplete natural language grammar. The grammar so far has the productions shown as Background in Fig. 1 for the non-terminals S (Sentence) and NP (Noun Phrase). The Example sentence cannot be explained by the Background knowledge, but can be explained by the Hypothesis in Fig. 1. Note that the Example is of predicate S, while the Hypothesis is of predicate NP. This contrasts with the usual Machine Learning setting of "Observation Predicate Learning" (OPL) in which examples and hypotheses define the same predicate. The simplified example in Fig. 1 is typical of the situation in grammar learning.

A non-OPL setting is natural for many problems involved in scientific discovery. For instance, at present in functional genomics (the determination of the function of genes from their gene sequence), the metabolic pathways of cells are being progressively determined. For a particular organism, such as yeast,

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 130–146, 2000.

	Grammar	Functional Genomics
Background	S → NP VP NP → DET NOUN	pheno_effect(Gene,Growth_medium) :- ... codes(Gene,Enzyme) ...
Example	S → the nasty man hit the dog	pheno_effect(gene,growth_medium).
Hypothesis	NP → DET ADJ NOUN	codes(gene,enzyme).

Fig. 1. Background knowledge, example and hypothesis involved in augmenting an incomplete **Grammar** and in a **Functional genomics** setting. "pheno_effect" is a shortening of "phenotypic_effect" (see Table 1).

it is possible to represent existing knowledge of metabolic pathways as a logic program (see Fig. 1). New experimental observation can then be explained by hypotheses such as the one shown in Fig. 1. Note once more the contrast with OPL.

The formation of hypotheses such as those shown in Fig. 1 has been intensively investigated within Abductive Logic Programming [7] (ALP). However, unlike the case in ILP, the hypotheses sought in ALP are typically not universally quantified laws. In this paper we describe an approach to non-OPL called Theory Completion using Inverse Entailment (TCIE). TCIE is based on inverse entailment [8] and is implemented within Progol5.0[1]. TCIE suffers from limitations of inverse entailment described by Yamamoto [20]. Augmentations of inverse entailment which address these limitations have been suggested in [9, 4], though these have not yet been implemented in Progol5.0. However, experiments in this paper show that good performance can be achieved using TCIE.

The paper has the following structure. Section 2 introduces TCIE as the special case of inverse entailment in which the construction of the bottom clause involves the derivation of negative ground instances from the background knowledge. In Section 2.1 it is shown how these negative instances can be derived using contra-positives introduced into the background knowledge in a fashion similar to that employed in Stickel's Prolog Technology Theorem Prover (PTTP). The multi-predicate search implemented in Progol5.0 is given in Section 2.3. Experiments are described which involve applying Progol5.0 to relearning a number grammar (Section 3) and missing enzymes in metabolic pathways (Section 4). A comparison with related work is given in Section 5. Section 6 concludes and describes future research.

2 TCIE

In the standard setting for ILP the learner is provided with logic programs describing background knowledge B and examples E and is required to find a consistent hypothesis H such that the following holds.

[1] The C source code for Progol5.0 and the datasets used for the experiments in this paper can be downloaded from ftp://ftp.cs.york.ac.uk/progol5.0 .

$$B \wedge H \models E \tag{1}$$

Inverse entailment [8] is based on the observation that (1) is equivalent for all B, H and E to the following.

$$B \wedge \overline{E} \models \overline{H} \tag{2}$$

Assuming E is a single example we can add its negation to B, deductively derive a finite conjunction of ground facts $\overline{\bot(B,E)}$, and then construct hypotheses H which subsume $\bot(B,E)$. The following non-OPL example of Inverse Entailment comes from [8].

Example 1. **Non-OPL example.**

$$\begin{aligned}
B &= \left\{ \begin{array}{l} \text{hasbeak}(X) \leftarrow \text{bird}(X) \\ \text{bird}(X) \leftarrow \text{vulture}(X) \end{array} \right\} \\
E &= \text{hasbeak}(\text{tweety}) \leftarrow \\
\overline{E} &= \leftarrow \text{hasbeak}(\text{tweety}) \\
\bot(B,E) &= \{\text{hasbeak}(\text{tweety}), \text{bird}(\text{tweety}), \text{vulture}(\text{tweety})\} \\
H_1 &= \text{bird}(\text{tweety}) \leftarrow \\
H_2 &= \text{bird}(X) \leftarrow \\
H_3 &= \text{vulture}(\text{tweety}) \leftarrow \\
H_4 &= \text{vulture}(X) \leftarrow
\end{aligned}$$

H_1, H_2, H_3, H_4 are potential hypotheses.

Yamamoto [20] has shown that this approach only derives clauses H for which H subsumes E relative to background knowledge B (see Plotkin [11] for the definition of relative subsumption). Relative subsumption is strictly weaker than the form of entailment shown in (1). In relative subsumption H can be used at most once in the derivation of E. This rules out, among other things, recursive applications of H.

Irrespective of its limitations, the non-OPL form of inverse entailment shown in Example 1 is not straightforward to implement using a Prolog interpreter. The problem stems from the need to derive negative ground literals (such as $\overline{\text{bird}(\text{tweety})}$) in the construction of $\bot(B,E)$. The following sections show how a Prolog implementation of non-OPL inverse entailment can be achieved based on an adaptation of ideas in Stickel's Prolog Technology Theorem Prover (PTTP).

2.1 PTTP and Contra-Positives

Stickel's PTTP [17] provides a method of applying a standard Prolog interpreter to theorem proving with arbitrary non-definite clauses. This is achieved by a number of transformations including the construction of contra-positives. Thus a clause with n atoms is stored as n different rules, a technique known as *locking*.

For example, $a \leftarrow b, c$ is also stored as $\overline{b} \leftarrow \overline{a}, c$ and $\overline{c} \leftarrow b, \overline{a}$. The effect of negated atomic formulae is achieved within the Prolog context by using extra predicate symbols. Thus $\overline{p(X)}$ is implemented as $non\_p(X)$. The following shows locking applied to $B \wedge \overline{E}$ from Example 1.

Example 2. **Non-OPL example revisited.**

$$B \wedge \overline{E} = \left\{ \begin{array}{l} \text{hasbeak}(X) \leftarrow \text{bird}(X) \\ \text{bird}(X) \leftarrow \text{vulture}(X) \\ \text{non\_bird}(X) \leftarrow \text{non\_hasbeak}(X) \\ \text{non\_vulture}(X) \leftarrow \text{non\_bird}(X) \\ \text{non\_hasbeak(tweety)} \leftarrow \end{array} \right\}$$

$$\overline{\bot(B,E)} = \{\text{non\_hasbeak(tweety)}, \text{non\_bird(tweety)}, \text{non\_vulture(tweety)}\}$$

$$\bot(B,E) = \{\text{hasbeak(tweety)}, \text{bird(tweety)}, \text{vulture(tweety)}\}$$

Clearly each ground atom in $\overline{\bot(B,E)}$ will succeed as a goal to a Prolog interpreter given the transformed $B \wedge \overline{E}$ in this example. The clause $\bot(B,E)$ is then formed by simply removing the prefix "non_".

2.2 Mode Declarations

Within Progol [8] the user indicates the language within which hypotheses are to be constructed using mode declarations. Mode declarations come in two forms: *modeh* statements indicate the predicates to be used in the head of hypothesised clauses and *modeb* statements indicate predicates to be allowed in the body. A mode declaration such as

```
:- modeb(1,p(+a,-b,#c))?
```

states that predicate 'p' has three arguments and will succeed with '1' answer substitution when called with the first argument ('+' represents input variable) bound with a term of type 'a'. It will return terms of type 'b' ('-' represents output variable) and 'c' ('#' represents constant) as its second and third arguments. The returned last argument will be used as a ground constant in the hypothesis.

Owing to the OPL assumption in previous versions of Progol, mode declarations not only define the hypothesis language but also define the separation between examples and background knowledge. Thus modeh statements indicate which predicates represent examples while modeb statements indicate which predicates represent background knowledge. In the non-OPL context of Progol5.0 modeh and modeb statements are still used to define the hypothesis language. Additional statements indicate which predicates are "observable". Observable predicates are those which represent examples. Fig. 2 shows the calling diagram and Progol5.0 declarations for the Grammar example of Fig. 1 and the Tweety example of Examples 1 and 2. Note that in the case of the Grammar, it is not necessary to construct contra-positive definitions for each predicate in the calling diagram. It is only necessary to do so for those on the paths in the calling diagram between "modeh" (hypothesis) predicates and "E" (observable) predicates.

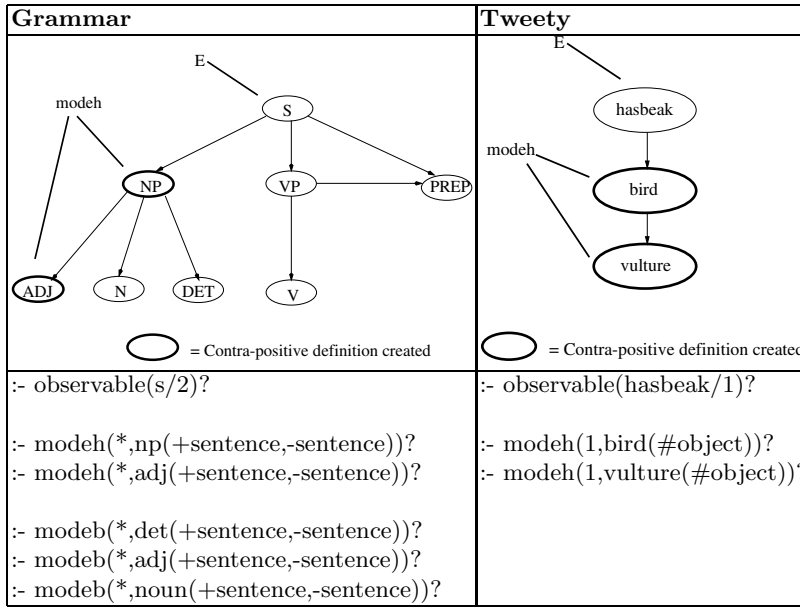


Fig. 2. Calling diagram and Progol5.0 declarations for **Grammar** from Fig. 1 and **Tweety** in Examples 1 and 2.

2.3 Multi-predicate Search Algorithm

As indicated in the Grammar example shown in Fig. 2 Progol5.0 can be given multiple modeh declarations. Progol5.0 uses a standard covering algorithm where each example is generalised using a multi-predicate search. This search is carried out, over all the predicates associated with modeh declarations, to find the hypothesis which covers the given example with maximal information compression. Compression is calculated using the Minimal Description Length function

$$f = p - (c + n)$$

where p is the number of positive examples covered among the observable predicates, c is the number of atoms in the body of the hypothesised clause and n is the number of negative examples covered by the hypothesised clause. The search related to each example to be generalised is carried out by the set of related procedures given in Fig. 3.

The underlying refinement graph search is based on the Progol procedure described in [8] (see Appendix A) modified to calculate compression over the observable predicates.

```
MultiPredicateSearch(ModehPredicates,Example)
    BestHypothesis=NULL              % Best hypothesised clause so far
    Max=0                            % Maximum Compression achieved
    For each P in ModehPredicates    % Maximise over all modeh predicates
        SinglePredicateSearch(P,Example,BestHypothesis,Max)
    End for each
    If Max>0 then return BestHypothesis

SinglePredicateSearch(P,Example,BestHypothesis,Max)
    Ms is Modehs(P)                  % Find all modeh declarations for P
    MakeContrapositives(P)           % Contra-positives of the form 'non_P'
    As are NegAtomsOf Ms             % Goals of the form "non_P"
    Derive ground atomic Start-set from calling As
    Obs are the observable predicates
    For each s in Start-set
        Let s' be s with 'non_' prefix removed.
        Let e be (s' ← Body(Example))
                                     % Example may be non-unit clause
        Construct ⊥_i from e         % See Section B.1
        Retract Example
        Find C, EmptyClause ⪯ C ⪯ ⊥_i with
            maximum compression Comp wrt Obs
                                     % See Section B.2
        If Comp > Max then
            Max = Comp
            BestHypothesis = C
        Assert Example
    End for each
```

Fig. 3. Multi-predicate search algorithm

3 Number Grammar Experiment

3.1 Materials

The experiment in this section involves learning a well-defined fragment of natural language, that is the translation of number phrases into their numerical form. For instance, the grammar translates the English phrase "three hundred and twenty-five" to the expression 3*100+2*10+5, or simply 325. Note that this is a special case of the general problem of translation of syntax to semantics. A complete Definite Clause Grammar (DCG) for the numbers 1-9999 is shown in Fig. 4.

3.2 Method

Note that the grammar in Fig. 4 has a hierarchically description similar to that of Fig. 1. The aim of the experiment is to determine whether Progol5.0 can

```
wordnum(A,[],T) :- tenu(A,[],T).
wordnum(A,[],T) :- word100(A,[],T).
wordnum(A,[],T) :- word1000(A,[],T).

word1000(A,[],T) :- thou(A,[],T).
word1000(A,B,T+N) :- thou(A,[and|C],T), tenu(C,B,N).
word1000(A,B,T+H) :- thou(A,C,T), word100(C,B,H).

thou([D,thousand|R],R,T*1000) :- digit(D,T).

word100(A,[],H) :- hun(A,[],H).
word100(A,B,H+T) :- hun(A,[and|C],H), tenu(C,B,T).

hun([D,hundred|R],R,H*100) :- digit(D,H).

tenu([D],[],N) :- digit(D,N).
tenu([ten],[],10).  tenu([eleven],[],11).  tenu([twelve],[],12).
tenu([thirteen],[],13).  tenu([fourteen],[],14).  tenu([fifteen],[],15).
tenu([sixteen],[],16).  tenu([seventeen],[],17).  tenu([eighteen],[],18).
tenu([nineteen],[],19).

tenu([T],[],N) :- tenmult(T,N).
tenu([T,D],[],N+M) :- tenmult(T,N), digit(D,M).

digit(one,1).  digit(two,2).  digit(three,3).  digit(four,4).
digit(five,5).  digit(six,6).  digit(seven,7).  digit(eight,8).
digit(nine,9).

tenmult(twenty,20).  tenmult(thirty,30).  tenmult(forty,40).
tenmult(fifty,50).  tenmult(sixty,60).  tenmult(seventy,70).
tenmult(eighty,80).  tenmult(ninety,90).
```

Fig. 4. Grammar for translating English number phrases to numbers.

recover performance of the complete theory in Fig. 4 after a randomly chosen subset of clauses from throughout the theory is removed.

The complete theory shown in Fig. 4 has 40 clauses. Progol5.0 was applied to learning from 100 randomly chosen examples with background knowledge consisting of partial grammars in which randomly chosen subsets of size 5, 10, 15 and 20 clauses were left out. For each size 10 randomly chosen left-out subsets were chosen and the results were averaged. Performance was measured on the complete set of 9999 examples.

3.3 Results

Fig. 5 shows the results, indicating the predictive accuracy measured both before and after relearning using Progol5.0. Error bars indicate the standard deviation

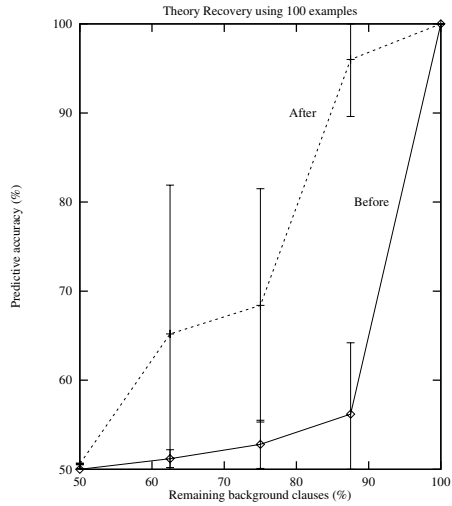


Fig. 5. Results of experiments on number grammar

in the predictive accuracy. Each experiment took around 0.1 seconds to run on a Dell 7000 Intel 686 laptop.

3.4 Discussion

The results in Fig. 5 indicate that substantial recovery of performance of a relatively complex grammar can be achieved when up to half of the grammar productions are deleted. However, when half or more of the clauses are deleted recovery fails. The reason for this appears to be related to the incompleteness of the approach discussed in [20]. That is, when multiple clauses are needed to complete a proof Progol5.0 fails to be able to reconstruct them all.

4 Functional Genomics Experiment

4.1 Context of Experiment

Genomic data is now being obtained on an industrial scale. The complete genomes of around a dozen micro-organisms have been sequenced. The genomes of about another 50 organisms are in the process of being sequenced and the completion of the sequencing of the human genome is imminent. The analysis of this data needs to become as industrialised as the methods for obtaining it.

The focus of genome research is moving to the problem of identifying the biological functions of genes. This is known as *functional genomics*. The problem is important because nothing is known about the function of between 30-60% of all new genes identified from sequencing [3, 5, 10]. Functional genomics is recognised as central to a deeper understanding of biology, and the future

exploitation of biology in medicine, agriculture, and biotechnology in general. Functional genomics is an appropriate application to test TCIE because: a) relational representations are appropriate for the problem; b) it is only possible to make experimental observations about concepts which are related to the target concept, as opposed to the target concept itself.

Metabolism and Growth Experiments One approach to functional genomics involves growth experiments. These involve feeding a micro-organism different mixtures of nutrients known as *growth media* and measuring the resulting growth. Cells of the micro-organism import molecules from a growth medium and convert them to molecules which are essential for growth via pathways of chemical reactions known as *metabolic pathways*. Each chemical reaction is catalysed by an enzyme. Some of these enzymes are known and others are not. The genes which code for the latter are genes whose function is not known.

To find out what the function of a gene is, mutants of a particular micro-organism are grown which do not contain the gene in question. This process is referred to as *gene deletion*. The effect of not having the gene can then be compared with a control i.e. the wild form of the organism which does not have the mutation. To make the effect of the mutation observable it is necessary to feed samples of the mutant with different growth media. The observable characteristics of an organism are collectively referred to as its *phenotype*. A phenotypic effect is a difference between the phenotype of the wild strain of an organism and the phenotype of one of its mutants.

4.2 Experimental Materials

The Functional Genomics Model In this experiment we simulate the effect of single-gene-deletion growth experiments using the abstract, highly simplified model of a cell shown in Fig. 6. In Computer Science terms, Fig. 6 may be viewed as a graph in which nodes represent molecules, arcs represent chemical reactions, labels of arcs represent enzymes which catatyse particular reactions and paths correspond to metabolic pathways. Although the model is highly simplified it is worthy of study because it has some of the characteristics of the functional genomics domain.

Table 1 lists part of the logic program which represents the functional genomics model. A growth medium is a combination of growth nutrients. Thus there are just seven growth media because there are just three nutrients. The program also contains:

- one `enzyme(enzyme, molecule_in, molecule_out)` fact for each enzyme in the model i.e. for each arc in the graph shown in Fig 6;
- `codes(gene, enzyme)` facts representing a one-to-one mapping of genes to enzymes.

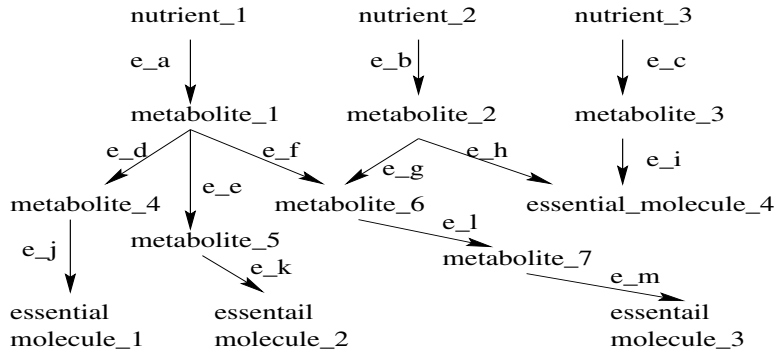


Fig. 6. Abstract, highly simplified model of a cell.

```
phenotypic_effect(Gene, Growth_medium):-
     nutrient_in(Nutrient, Growth_medium),
     metabolic_path(Nutrient, Mi),
     enzyme(E, Mi, Mj),
     codes(Gene, E),
     metabolic_path(Mj, Mn),
     essential_molecule(Mn),
     not(path_without_E(Growth_medium, Mn, E)).

nutrient_in(Nutrient, Growth_medium):- element(Nutrient, Growth_medium).

metabolic_path(A, A).
metabolic_path(A, B):- enzyme(_, A, B).
metabolic_path(A, B):- enzyme(_, A, X), metabolic_path(X, B).

path_without_E(Growth_medium, Mn, E):-
     nutrient_in(Nutrient, Growth_medium),
     path_without_E(Nutrient, Mn, E).
path_without_E(A,A,_).
path_without_E(A,B,E):- enzyme(E2,A,B),not(E=E2).
path_without_E(A,B,E):- enzyme(E2,A,X),not(E=E2),path_without_E(X,B,E).

essential_molecule(ess_mol_1).       essential_molecule(ess_mol_2).
essential_molecule(ess_mol_3).       essential_molecule(ess_mol_4).
```

Table 1. Part of the logic program which represents the functional genomics model.

Table 2. Experimental Method

for j **in** (23, 45, 68, 91)
  a training set was created by selecting j examples at random;
  for k **in** (0, 1, 4, 7, 10, 12, 13)
    for i **in** 1 **to** 10 **do**
      – k `codes/2` facts were selected at random;
      – the other $13 - k$ `codes/2` facts were removed from the model;
      – the performance of the resulting incomplete model was measured;
      – Logical Back Propagation was applied to the training set and the incomplete model;
      – the performance of the updated model was measured.
    end
  end
end

The Example Set Examples were represented by positive or negative instances of the predicate `phenotypic_effect(gene, growth_medium)`. For example `phenotypic_effect(g1, [nutrient_1, nutrient_2])` represents the fact that a phenotypic effect is observed if a mutant strain of an organism is created by removing the gene `g1` and this mutant is fed the growth medium which contains nutrients 1 and 2. There are 91 such examples since there are seven examples for each of the 13 genes in the model: there are only seven possible growth media. The class of each example (positive or negative) was deduced from the complete model. There are 45 positives and 46 negatives.

4.3 Experimental Method

Table 2 summarises the experimental method. The performance measure used was predictive accuracy on the complete distribution of the observable predicate i.e. `phenotypic_effect(gene, growth_medium)`. The effect of the inner-most loop is to measure pre-learning and post-learning performance ten times for every possible pair of j and k values shown in Table 2; the purpose of this repetition was to allow the subsequent calculation of the mean accuracy and standard deviation for each pair of j and k values.

It was not possible to use a purely inductive approach to regenerate the `codes/2` facts from the training data because `codes/2` is below `phenotypic_effect/2` in the calling diagram. CProgol was instructed to abduce `codes/2` facts as follows.

```
:- modeh(1,codes(#gene,#enzyme))?
:- observable(phenotypic_effect/2)?
```

During training, Progol was instructed to assume that `codes(gene, enzyme)` is a one-to-one mapping by placing the following constraints in the learning file.

```
:- codes(Gene, Enzyme1), codes(Gene, Enzyme2), not (Enzyme1 = Enzyme2).
:- codes(Gene1, Enzyme), codes(Gene2, Enzyme), not (Gene1 = Gene2).
```

4.4 Results and Analysis

Fig. 7 shows a plot of the results in the form of four learning curves. The results show that applying TCIE to a version of the model which has been made incomplete by removing some of `codes/2` facts leads to a recovery in performance, regardless of how many facts were removed or the size of the training set. The more `codes/2` facts that are removed, the greater the recovery. The larger the training set, the greater the recovery.

Running times for the experiments in Fig. 7 were typically under 6 seconds on a Silicon Graphics O2 workstation.

5 Comparison with Related Work

TCIE fits within the general scheme of theory refinement described in [19]. Within this scheme TCIE is a form of theory revision based on generalising (completing) revisions. Earlier systems with comparable aims include MIS [16] CLINT [12, 13], RUTH [1], AUDREY [18] and BORDA [6] (see also [15] and [14]). We would argue that though similar in spirit to many of these earlier systems, TCIE as implemented in Progol5.0, has advantages related to the inheritance of efficient pruning, logical constraints and the general Progol framework from earlier versions of Progol. Progol5.0's efficiency is indicated by the fact that running times for experiments in this paper were typically under 6 second on a standard workstation.

Many of the basic mechanisms used throughout theory revision systems are based on abductive operators from logic programming. In [2] Dimopoulos and Kakas compared abductive and inductive forms of reasoning. They note that ILP and abduction share the following relationship among background theory T, hypothesis H and observation O.

$$T, H \models O$$

They summarise the main difference between abduction and induction as follows.

> Whilst in abduction the addition of H to T just adds missing facts related to the particular observations in induction H introduces new general relations in T.

TCIE incorporates a form of abduction based on the use of contra-positives. This is used to derive the set of ground atoms referred to as the Start-set in the algorithm in Fig. 3. However, TCIE then goes on to generalise these facts to find new general relations. We would therefore argue that TCIE is a form of induction which incorporates abductive reasoning.

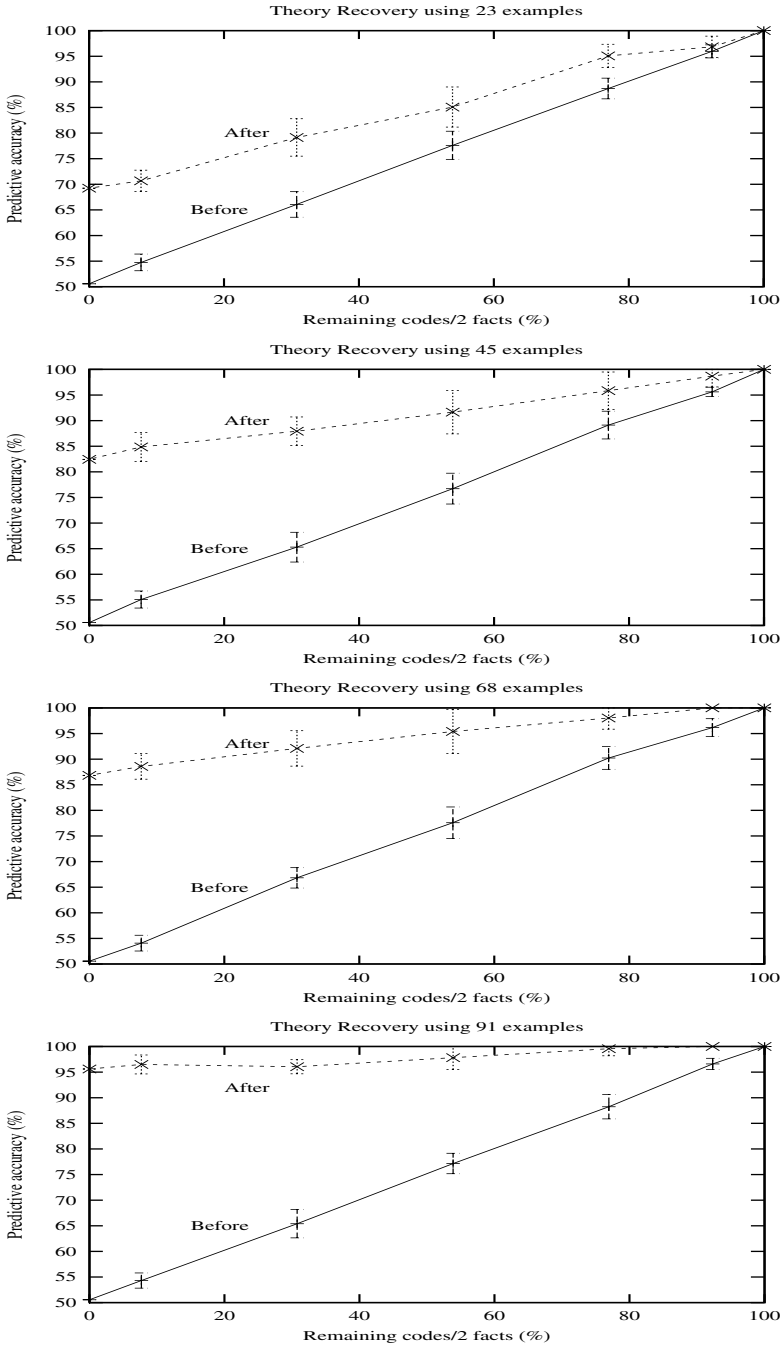


Fig. 7. Learning curves for the model of functional genomics.

6 Conclusions and Further Work

In this paper we describe the non-OPL setting for ILP. Here the predicates involved in observations are distinct from those in the head of hypothesised clauses. It is shown how Inverse Entailment has been used to implement TCIE within Progol5.0. Progol5.0 is then applied in two experiments to datasets in which random subsets of the complete theory are deleted. The results indicate that Progol5.0 is capable of recovering predictive accuracy to a substantial degree, even when large sections of the background knowledge have been deleted.

However, it was noted in the discussion in Section 3.4 that the incompleteness described by Yamamoto [20] limits the performance of Progol5.0. This incompleteness stems from multiple uses of the hypothesised clause within the derivation of an example. The performance limitation might come as some surprise in that this incompleteness limitation has often been assumed to be related to learning of recursive theories which necessarily require recursive hypotheses to be used more than once. The grammar shown in Fig. 3 is non-recursive. However, consider parsing the phrase "two hundred and two". This parse uses the ground unit clause digit(two,2) twice. Therefore if this ground unit clause is missing from the background theory then the incompleteness noted by Yamamoto will prevent its being relearned. Thus methods to circumvent this form of incompleteness are important for future research.

The datasets used in this paper differ substantially from those used in the more familiar OPL setting, in which there is only one predicate to be learned. We believe that the novel testing methodology used in the experimental sections of this paper are important contribution. Moreover, we believe that the non-OPL setting should be of increasing interest in applications of ILP.

Acknowledgements

The authors would like to thank Akihiro Yamamoto, Koichi Furukawa, Antonis Kakas, Stefan Wrobal and Luc De Raedt for discussions about Abduction and inverse entailment. The first author would like to thank his wife wife Thirza and daughter Clare for their cheerful support during the writing of this paper. This work was supported partly by the Esprit RTD project "ALADIN' (project 28623), EPSRC grant "Closed Loop Machine Learning", BBSRC/EPSRC grant "Protein structure prediction - development and benchmarking of machine learning algorithms" and EPSRC ROPA grant "Machine Learning of Natural Language in a Computational Logic Framework".

References

[1] H. Ade, L. De Raedt, and M. Bruynooghe. Theory revision. In S. Muggleton, editor, *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 179–192, 1993.

[2] Y. Dimopoulos and A. Kakas. Abduction and inductive learning. In L. De Raedt, editor, *Proceedings of the Fifth Inductive Logic Programming Workshop (ILP95)*, pages 25–28, Leuven, Belgium, 1995. KU Leuven.

[3] B. Dujon. The yeast genome project - what did we learn? *Trends in Genetics*, 12:263–270, 1996.

[4] K. Furukawa. On the completion of the most specific hypothesis computation in inverse entailment for mutual recursion. In *Proceedings of Discovery Science '98*, LNAI 1532, pages 315–325, Berlin, 1998. Springer-Verlag.

[5] Goffeau, A. *et multi al.* Life with 6000 genes. *Science*, 274:546–567, 1996.

[6] K. Ito and A. Yamamoto. Finding hypotheses from examples by computing the least generlisation of bottom clauses. In S. Arikawa and H. Motoda, editors, *Proceedings of Discovery Science '98*, pages 303–314. Springer, Berlin, 1998. LNAI 1532.

[7] A.C. Kakas, R.A. Kowalski, and F. Toni. Abductive logic programming. *Journal of Logic and Computation*, 2, 1992.

[8] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[9] S. Muggleton. Completing inverse entailment. In C.D. Page, editor, *Proceedings of the Eighth International Workshop on Inductive Logic Programming (ILP-98)*, LNAI 1446, pages 245–249. Springer-Verlag, Berlin, 1998.

[10] S.G. Oliver. From DNA sequence to biological function. *Nature*, 379:597–600, 1996.

[11] G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6. Edinburgh University Press, 1971.

[12] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[13] L. De Raedt and M. Bruynooghe. Interactive concept-learning and constructive induction by analogy. *Machine Learning*, 8:107–150, 1992.

[14] L. De Raedt and N. Lavrac. Multiple predicate learning in two inductive logic programming settings. *Journal on Pure and Applied Logic*, 4(2):227–254, 1996.

[15] B. L. Richards and R. J. Mooney. Automated refinement of first-order Horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[16] E.Y. Shapiro. *Algorithmic program debugging.* MIT Press, 1983.

[17] M. Stickel. A Prolog technology theorem prover: implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, 4(4):353–380, 1988.

[18] J. Wogulis. Revising relational theories. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 462–466. Morgan Kaufmann, 1991.

[19] S. Wrobel. First-order theory refinement. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 14–33. IOS Press, Ohmsha, Amsterdam, 1995.

[20] A. Yamamoto. Which hypotheses can be found with inverse entailment? In N. Lavrač and S. Džeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, pages 296–308. Springer-Verlag, Berlin, 1997. LNAI 1297.

A Progol Algorithm

B Definition of Most-Specific Clause

Definition 1. Most-specific clause $\bot_i$. *Let h, i be natural numbers B be a set of Horn clauses, $e = a \leftarrow b_1, .., b_n$ be a definite clause, M be a set of mode declarations containing exactly one modeh m such that $a(m) \preceq a$ and $\bot$ be the most-specific (potentially infinite) definite clause such that $B \wedge \bot \wedge \overline{e} \vdash_h EmptyClause$. $\bot_i$ is the most-specific clause in $\mathcal{L}_i(M)$ such that $\bot_i \preceq \bot$.*

B.1 Construction of Most-Specific Clause

Algorithm 1 Algorithm for constructing $\bot_i$.

1. *Given natural numbers h, i, Horn clauses B, definite clause e and set of mode declarations M.*
2. *Let $k = 0$, $hash : Terms \rightarrow N$ be a hash function which uniquely maps terms to natural numbers, $\overline{e}$ be the clause normal form logic program $\overline{a} \wedge b_1 \wedge .. \wedge b_n$, $\bot_i = \langle\rangle$ and $InTerms = \emptyset$.*
3. *If there is no modeh in M such that $a(m) \preceq a$ then return $EmptyClause$. Otherwise let m be the first modeh declaration in M such that $a(m) \preceq a$ with substitution θ_h. Let a_h be a copy of $a(m)$ and for each v/t in θ_h if v corresponds to a #type in m then replace v in a_h by t otherwise replace v in a_h by v_k where $k = hash(t)$ and add v to $InTerms$ if v corresponds to +type. Add a_h to $\bot_i$.*
4. *If $k = i$ return $\bot_i$ else $k = k + 1$.*
5. *For each modeb m in M let $\{v_1, .., v_n\}$ be the variables of +type in $a(m)$ and $T(m) = T_1 \times .. \times T_n$ be a set of n-tuples of terms such that each T_i corresponds to the set of all terms of the type associated with v_i in m (term t is tested to be of a particular type by calling Prolog with type(t) as goal). For each $\langle t_1, .., t_n \rangle$ in $T(m)$ let a_b be a copy of $a(m)$ and $\theta = \{v_1/t_1, .., v_n/t_n\}$. If Prolog with depth-bound h succeeds on goal $a_b\theta$ with the set of answer substitutions Θ_b then for each θ_b in Θ_b and for each v/t in θ_b if v corresponds to a #type in m then replace v in a_b by t otherwise replace v in a_b by v_k where $k = hash(t)$ and add v to $InTerms$ if v corresponds to -type. Add $\overline{a_b}$ to $\bot_i$.*
6. *Goto step 4.*

B.2 *A**-like Algorithm for Finding Clause with Maximal Compression

Firstly we define some auxiliary functions used in Algorithm 2.

Definition 2. Auxiliary functions. *Let the examples E be a set of Horn clauses. Let $h, i, B, e, M, \bot_i$ be as in Definition 1. Let C be a clause, k be a natural number and θ be a substitution.*

$$d'(v) = \begin{cases} 0 & \textit{if there is no -type variable in the head of } \bot_i \\ 0 & \textit{if } v \textit{ is -type in the head of } \bot_i \\ \infty & \textit{if } v \textit{ is not in } \bot_i \\ (min_{u \in U_v} d'(u)) + 1 & \textit{otherwise} \end{cases}$$

where U_v are the -type variables in atoms in the body of C which contain +type occurrences of v. Below state s has the form $\langle C, \theta, k \rangle$. c is a user-defined parameter for the maximal clause body length. $|S|$ denotes the cardinality of any set S.

$$\begin{aligned} p_s &= |\{e : e \in E \textit{ and } B \wedge C \wedge \overline{e} \vdash_h \textit{EmptyClause}\}| \\ n_s &= |\{e : e \in E \textit{ and } B \wedge C \wedge e \vdash_h \textit{EmptyClause}\}| \\ c_s &= |C| - 1 \\ V_s &= \{v : u/v \in \theta \textit{ and } u \textit{ in body of } C\} \\ h_s &= min_{v \in V_s} d'(v) \\ g_s &= p_s - (c_s + h_s) \\ f_s &= g_s - n_s \end{aligned}$$

best(S) is a state $s \in S$ which has $c_s \leq c$ and for which there does not exist $s' \in S$ for which $f_{s'} > f_s$.

$$prune(s) = \begin{cases} true & \textit{if } n_s = 0 \textit{ and } f_s > 0 \\ true & \textit{if } g_s \leq 0 \\ true & \textit{if } c_s \geq c \\ false & \textit{otherwise} \end{cases}$$

$$terminated(S, S') = \begin{cases} true & \textit{if } s = best(S), n_s = 0, f_s > 0 \textit{ and} \\ & \textit{for each } s' \textit{ in } S' \textit{ it is the case that } f_s \geq g_{s'} \\ false & \textit{otherwise} \end{cases}$$

Algorithm 2 Algorithm for searching $EmptyClause \preceq C \preceq \bot_i$**.**

1. *Given $h, B, e, \bot_i$ as in Definition 7.*
2. *Let $Open = \{\langle EmptyClause, \emptyset, 1 \rangle\}$ and $Closed = \emptyset$.*
3. *Let $s = best(Open)$ and $Open = Open - \{s\}$.*
4. *Let $Closed = Closed \cup \{s\}$.*
5. *If $prune(s)$ goto 7.*
6. *Let $Open = (Open \cup \rho(s)) - Closed$.*
7. *If $terminated(Closed, Open)$ then return $best(Closed)$.*
8. *If $Open = \emptyset$ then print 'no compression' and return $\langle e, \emptyset, 1 \rangle$.*
9. *Goto 3.*

Solving Selection Problems Using Preference Relation Based on Bayesian Learning

Tomofumi Nakano and Nobuhiro Inuzuka

Nagoya Institute of Technology,
Gokiso-cho, Showa-ku, Nagoya 466-8555, Japan
{tnakano, inuzuka}@ics.nitech.ac.jp

Abstract. This paper defines a selection problem which selects an appropriate object from a set that is specified by parameters. We discuss inductive learning of selection problems and give a method combining inductive logic programming (ILP) and Bayesian learning. It induces a binary relation comparing likelihood of objects being selected. Our methods estimate probability of each choice by evaluating variance of an induced relation from an ideal binary relation. Bayesian learning combines a prior probability of objects and the estimated probability. By making several assumptions on probability estimation, we give several methods. The methods are applied to Part-of-Speech tagging.

1 Introduction

Inductive logic programming (ILP) methods have dealt with many classes of problems. We frequently meet a problem to choose an object from a set. Multi-class classification is also regarded as a typical selection problem, which chooses a class from a set of classes depending on some attributes or background knowledge. Part-of-speech (POS) tagging, whose purpose is to give a POS tag to each word in sentences, is also a selection problem. This case is more complex than multi-class classification in the point of sets from which an element is selected. The POS tagging chooses a POS tag from a different set depending on a word, while multi-class classification chooses a class always from the same set of classes. In the game tree search of chess-like games, to select an appropriate move is also a selection problem. In this case there are also different sets of moves of which a move is selected, depending on the current game state. The set is a set of legal moves of the state.

This paper discusses a way to learn selection inductively, in the framework of concept learning and Bayesian learning. As we discuss through the paper, when we deal with selection as a concept, the concept has to be biased. Because of this bias, we can infer plausible selection in spite of an incomplete result of learning.

We give probability models where incomplete results of learning are varied from an ideal selection concept, and use it to estimate the most plausible selection. The selection is defined by combining estimated probability from a learned relation and the prior probability. It is advantageous to estimate probability from

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 147–164, 2000.

relations induced by ILP, because the probability can be integrated with other statistical information. This merit motivates some researches[2,4,7,18]. They attempt to induce probability based on relational learning. In particular [2,7] induce probability distribution of unknown examples based on empirical distribution. Our methods give conditional probability of a hypothesis or selection as a biased concept without considering probability distribution of examples.

This paper proposes to use a binary relation which compares two objects to be selected. We used in papers[11,17] this idea to induce rules for appropriate moves of a chess-like game. J. D. Hirst et.al. also used a binary relation to estimate a chemical quantity using ILP. [12] takes a similar approach, which classifies each of comparison pairs (comparison among classes) and aggregates the results for multi-class problems based on the attribute-value representation. If we have a binary relation which compares objects, selecting the most plausible one can be reduced to ranking in the relation. To have a winner of round robin tournament, in the sense of the comparison relation, is a common idea and variations of this idea are investigated in [6]. We explain some of them from the Bayesian model.

We attempt to adopt selection methods to POS tagging according to the experiment by J.Cussens[1,3]. Learning using ILP seems suitable for such structural and complex domains like natural languages. The selection using binary relation representation allows us to use comparison of objects and their attributes. Accordingly we expect ILP methods to work effectively.

Section 2 defines selection problems and learning, and also gives two kinds of representation, unary and binary relation representation, of selection as concept learning. Section 3 discusses the Bayesian learning framework to solve the selection learning, and gives solutions for unary relation representation. Solutions for binary relation representation is treated in Section 4. Then, we explain experiments of POS tagging in Section 5 and conclude in Section 6.

2 Selection Problems and Selection Learning

2.1 Definitions

First of all we give a definition of selection as follows.

Definition 1 (selection) *A set X, called an* object set*, and a set Θ, called a* parameter space*, which can be decomposed into a certain set Θ_1, whose elements are called domain parameters, and another set Θ_2, i.e., $\Theta = \Theta_1 \times \Theta_2$, are given. A function*

$$f : \Theta_1 \to 2^X$$

called a domain function *is also given. The value of this function $f(i_{\Theta_1}(\theta))$ is denoted by X_θ, where i_{Θ_1} is the projection from Θ to Θ_1. Then,* selection s *defined over X, Θ and f is to choose an object $o \in X_\theta$ by given a parameter $\theta \in \Theta$, i.e., a function defined as*

$$\begin{aligned} s : \Theta &\to X \\ \theta &\mapsto o \in X_\theta \subseteq X. \end{aligned}$$

□

We call a problem to find selection a *selection problem.*

For example, let us imagine to find the first prize horse for each horse racing. In this case the object set is a set of all horses, and the parameter space consists of dates of races, conditions of field, weather, and so on. Dates should be used for a domain parameter and the domain function is a function which maps a date to a set of horses that participate in the race on the date, the set which is a subset of the object set. The selection in this case is to select a horse from the set of participating horses specified by dates, by using the other parameters.

Many examples of selection problems are found when the domain function is a constant function that always outputs X, i.e., $\forall\theta \in \Theta, X_\theta = X$. In this case selection is to choose an object $o \in X$ by given a parameter. Multi-class classification has this structure.

This paper discusses to solve selection problems by an inductive method. Accordingly we define a learning problem, *selection learning.*

Definition 2 (selection learning) *When X is a given object set and Θ is a parameter space with a domain parameter and a domain function,* selection learning *is to infer selection s defined over X and Θ from a set D,*

$$D \subseteq \{(\theta, o) \in \Theta \times X \mid o = s(\theta)\}. \tag{1}$$

The s is called target selection *and D is called a* training set. *The projection of D to the parameter space*

$$\Theta_D = \left\{\theta \in \Theta \;\middle|\; \exists o \in X, (\theta, o) \in D\right\}$$

is called training parameters. □

The rest of this section gives two frameworks to treat selection learning as concept learning. One is to restrict objects and to induce knowledge to classify objects to be chosen or not to be in unary representation. The other framework is to induce knowledge for preference which tells an object is preferred to another for the selection in binary representation.

2.2 Restriction Knowledge

An approach to treat selection is to reduce it to a concept that restricts objects. We define concepts c_θ indexed by a parameter $\theta \in \Theta$ as follows.

$$\begin{aligned} c_\theta : X_\theta &\rightarrow \{1, 0\} \\ o &\mapsto \begin{cases} 1 & \text{if } s(\theta) = o \\ 0 & \text{if } s(\theta) \neq o \end{cases} \end{aligned}$$

Accordingly a selection learning task is reduced to a concept learning task that infers parameterized concept c_θ from the following training examples D', which are transformed from D of Equation (1).

$$D' = \{(\theta, o, sign) \in \Theta \times X \times \{0, 1\} \mid \theta \in \Theta_D \wedge o \in X_\theta \wedge sign = c_\theta(o)\}$$

Examples with $sign = 1$/$sign = 0$ are positive/negative examples of the target.

The concept $c_\theta(o)$ for every θ takes 1 for exactly one object o in X_θ. Hence, we need to give bias for learning to infer such concepts. If we can not give such bias, an inferred concept is not transformed to selection. Nevertheless, an inferred concept for c_θ tells allowance of selecting objects for target selection. We call this representation using c_θ of selection *unary relation representation*.

2.3 Preference Knowledge

For another approach to reduce selection learning to concept learning, we consider the following parameterized binary concepts.

$$r_\theta : \quad X_\theta^2 \quad \to \quad \{1, 0\}$$

$$(o_1, o_2) \mapsto \begin{cases} 1 & \text{if } s(\theta) = o_1 \\ 0 & \text{if } s(\theta) = o_2 \text{ and } o_2 \neq o_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We understand that this relation r_θ represents comparison of preference or likelihood of objects for selection. We can deal with the original selection task as a concept learning with this binary concept for a target concept. Although it seems strange that the target concept is not defined completely, the answer of the selection task or the object to be chosen can be clearly defined as the maximum element in X_θ in the sense of a relation $\geq$ defined by $o \geq o' \leftrightarrow r_\theta(o, o')$.

The binary concept only expresses a relation between an object to be selected and others, while a relation extended to all of combination of objects is imaginable. We call this representation of the selection using r_θ a *binary preference relation representation*.

For concept learning of the target concept r_θ, the following D'' is a training set transformed from D.

$$D'' = \left\{ \begin{array}{l} (\theta, o, o', sign) \\ \in \Theta \times X^2 \times \{0, 1\} \end{array} \;\middle|\; \begin{array}{l} \theta \in \Theta_D \wedge o \in X_\theta \\ \wedge o' \in X_\theta \wedge sign = r_\theta(o, o') \end{array} \right\}$$

In D'' examples with $sign = 1$/$sign = 0$ are used as positive/negative examples of r_θ. Examples with $sign =$ undefined should be ignored in learning.

We have defined selection problems and given two methods to transform them to concept learning tasks that target c_θ or r_θ. Let us assume that we make prediction of c_θ or r_θ from their training examples D' or D'', respectively, by using some induction methods. The following two sections discuss to predict an answer of selection learning from induced concepts for c_θ or r_θ in the Bayesian learning framework.

3 The Selection of Objects Using Induced Knowledge Based on Bayesian Learning

After we review the framework of maximum a posteriori (MAP) hypothesis, we adopt it to predict selection from induced concept for c_θ. In the rest of this paper we denote a concept inferred for the target c_θ or r_θ by R.

3.1 Maximum a Posteriori Object

According to the Bayesian theorem, we have the following equation

$$p(h|D) = \frac{p(h) \cdot p(D|h)}{p(D)},$$

where h denotes the event that a hypothesis, denoted also by h, is true, and D denotes the event that a set of data, denoted also by D, is observed.

Then, the most probable hypothesis in H or the *maximum a posteriori* (*MAP*) *hypothesis* under the observation of data D is the following hypothesis h_{MAP}.

$$h_{\mathrm{MAP}} = \underset{h \in H}{\operatorname{argmax}}\, p(h|D) = \underset{h \in H}{\operatorname{argmax}}\, p(h) \cdot p(D|h)$$

Let us apply this equation to our selection problem. Each object in X corresponds to a hypothesis h, because it is what we want to predict. On the other hand, if we encapsulate the induction process of R, we can regard the presence of R as an observation. As a result, we have

$$o_{\mathrm{MAP}} = \underset{o \in X}{\operatorname{argmax}}\, p(o) \cdot p(R|o), \tag{2}$$

where o_{MAP} is the object that is maximally probable to be chosen, and $p(o)$ is the prior probability that o will be chosen. $p(R|o)$ is the posterior probability that a relation R will be appeared under the condition that o is the true choice.

In order to use Equation (2) we will think of the posterior probability $p(R|o)$. In the rest of this section we give an estimation of $p(R|o)$ when we take concept learning based on c_θ. We leave the case of r_θ to the next section.

3.2 The Selection Based on a Unary Relation

Let a set $X = \{o_1, \cdots, o_n\}$ be an object set, and R be an inferred relation for the target c_θ. Of course $R \subseteq X$. If an object $o \in X$ is the object of the correct choice, the ideal unary relation based on c_θ is $\widehat{R} = \{o\}$. For an assumption to estimate the probability $p(R|o)$, we give a constant probability ρ to the event that an object to be selected is in the induced relation R and also to the event that an object not to be selected is not in R. These events happen $n - |R \triangle \widehat{R}| = n - |R \triangle \{o\}|$ times independently, where $A \triangle B = (A - B) \cup (B - A)$. The independence is also assumed. The event that an object to be selected is not in R and the event that an object not to be selected is in R has the probability $1 - \rho$. These events happen $|R \triangle \widehat{R}| = |R \triangle \{o\}|$ times. Then, when we have a c_θ-base prediction, the probability of a unary relation R predicted under condition that an object o is true selection is

$$p(R|o) = (1-\rho)^{|R \triangle \{o\}|} \cdot \rho^{n - |R \triangle \{o\}|} = \left(\frac{1-\rho}{\rho}\right)^{|R \triangle \{o\}|} \cdot \rho^n. \tag{3}$$

We should note that in Equation (3) the probability ρ can be estimated by the accuracy of R, i.e.,

$$\rho = \frac{(\text{the number of correctly classified positive and negative data by } R)}{(\text{the number of all positive and negative data})}.$$

In order to have this estimator of ρ we need to test R using a separated example set from training examples that are used to induce R and also from the test examples that will be finally used for evaluation. We call the examples as *parameter estimation examples*.

As a result by substituting Equation (2) by Equation (3) we yield the most probable object.

$$o_{\mathrm{MAP}} = \operatorname*{argmax}_{o \in X} p(o) \cdot \left(\frac{1-\rho}{\rho}\right)^{|R \triangle \{o\}|} \cdot \rho^n = \operatorname*{argmax}_{o \in X} p(o) \cdot \left(\frac{1-\rho}{\rho}\right)^{|R \triangle \{o\}|} \quad (4)$$

We call selection using this equation selection based on the *unary relation representation* (**URR**).

Let $|R| = k$. Then $|R \triangle \{o\}| = |R - \{o\}| = k - 1$ if $o \in R$. Otherwise, $|R \triangle \{o\}| = k + 1$. Hence, the probability $p(R|o)$ is reduced to

$$p(R|o) = \begin{cases} \left(\frac{1-\rho}{\rho}\right)^{k-1} \cdot \rho^n & \text{if } o \in R \\ \left(\frac{1-\rho}{\rho}\right)^{k+1} \cdot \rho^n & \text{if } o \notin R. \end{cases}$$

Every object in R has the equal posterior probability and every object not in R has another smaller equal probability. Then, if we assume that

$$\left(\frac{1-\rho}{\rho}\right)^2 \max_{o \in X} p(o) < \min_{o \in X} p(o),$$

o_{MAP} is defined as

$$o_{\mathrm{MAP}} = \begin{cases} \operatorname*{argmax}_{o \in X \cap R} p(o) & \text{if } R \neq \emptyset \\ \operatorname*{argmax}_{o \in X} p(o) & \text{if } R = \emptyset. \end{cases} \quad (5)$$

The most plausible object is one with the maximum prior probability $p(o)$ within objects allowed by R. When there is no object allowed by R, the object with maximum $p(o)$ is the most plausible. As we describe in Section 5, Cussens reported an experiment of applying an ILP method to detect POS tags of words in English sentences. The selection of tags in his experiment obeys Equation (5). Accordingly we refer to this selection method as **Cussens'**.

In Equation (3) we assumed the probability of agreeing the membership of each object in the ideal relation c_θ and that of an induced relation R to be a constant. On the other hand, if we give different two probabilities ρ_+ and ρ_- to events that an object in c_θ is in R and to events that an object not in c_θ is not in R, respectively, the equation becomes

$$p(R|o) = (1-\rho_+)^{|\{o\}-R|} \cdot \rho_+^{1-|\{o\}-R|} \cdot (1-\rho_-)^{|R-\{o\}|} \cdot \rho_-^{(n-1)-|R-\{o\}|}. \quad (6)$$

According to this equation the o_{MAP} object is defined as

$$o_{\mathrm{MAP}} = \underset{o \in X}{\operatorname{argmax}}\, p(o) \cdot \left(\frac{1-\rho_+}{\rho_+}\right)^{|\{o\}-R|} \cdot \left(\frac{1-\rho_-}{\rho_-}\right)^{|R-\{o\}|}.$$

We refer to the selection using this equation as **URR2**.

The probability ρ_+ can be estimated by the sensitivity or

$$\rho_+ = \frac{(\text{the number of correctly classified positive data by } R)}{(\text{the number of all positive data})},$$

and ρ_- is estimated by the specificity or

$$\rho_- = \frac{(\text{the number of correctly classified negative data by } R)}{(\text{the number of all negative data})}.$$

4 Maximum Posteriori Selection Based on Binary Preference Relation

Section 2.3 gave the other transformation of selection learning to concept learning. It treats selection as a binary relation. By extending the discussion of the previous section, we give the posterior probability of presence of an inferred binary relation. We can imagine two different assumptions on the ideal binary relations to be induced and these give different results.

When we deal with many objects using all possible pairs of them, a winning point and a lost point on their round robin tournaments are common consideration. Our discussion gives understanding relationship between the maximum posteriori selection and a winner in round robin tournaments.

4.1 Total Order Assumption

The first idea for calculating the posterior probability assumes that the ideal preference relation, which is guessed as R, is a total order relation on X and the object to be chosen is the maximum element on this total order relation. We denote this ideal total order by $\widehat{R}$. Here this assumption is called *total order assumption*.

We also assume that for each pair $(o_i, o_j) \in X^2$ the membership of the pair in R and that in $\widehat{R}$ meet with a constant probability, say φ, and differ with $1-\varphi$.

If we denote an event that the ideal relation is $\widehat{R}$ also by $\widehat{R}$, we can get[1]

$$p(R|\widehat{R}) = (1-\varphi)^{|R\triangle\widehat{R}|} \cdot \varphi^{n^2-|R\triangle\widehat{R}|} = \left(\frac{1-\varphi}{\varphi}\right)^{|R\triangle\widehat{R}|} \cdot \varphi^{n^2}. \tag{7}$$

[1] The ideal relation is a total order and is reflexive, and so it is natural that the learned relation R is constrained to be reflexive. In this case R and $\widehat{R}$ always meet in their diagonal elements. However, it is difficult to give this constraint in a learning process. Equation (7) does not assume this constraint.

For an object $o \in X$ there are $(n-1)!$ total order relations in which the maximum object is o. Let T_o denote the set of such total order relations. Then, $p(R|o)$ is the sum of $p(R|\widehat{R})$ for all $\widehat{R} \in T_o$, i.e.,

$$p(R|o) = \sum_{\widehat{R} \in T_o} p(R|\widehat{R}) = \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|} \cdot \varphi^{n^2}. \tag{8}$$

We can use the accuracy of the induced relation $\widehat{R}$ to classify pairs in X^2 as an estimator of φ, as we used the accuracy of the inferred relation for c_θ for ρ in Section 3.2. This estimator needs to be prepared from a separate set of parameter estimation examples from test examples and also from training examples of induction.

For this probability $p(R|o)$ we can derive the following theorem.

Theorem 1 *When a relation R is a total order on X with the maximum element $o^* \in X$ and $\varphi > 0.5$, then*

$$p(R|o^*) > p(R|o) \quad \textit{for all } o \in X, o \neq o^*,$$

where $p(R|o)$ is defined by Equation (8).

(See Appendix for proof) □

Equations (2) and (8) yield

$$o_{\mathrm{MAP}} = \operatorname*{argmax}_{o \in X} p(o) \cdot \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|} \cdot \varphi^{n^2} = \operatorname*{argmax}_{o \in X} p(o) \cdot \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|}. \tag{9}$$

We use Equation (9) as a selection method, which we call selection based on *total order assumption* (**TOA**).

When we have no prior knowledge on $p(o)$ and we have to assign it a constant probability, we get

$$o_{\mathrm{MAP}} = \operatorname*{argmax}_{o \in X} \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|}.$$

Furthermore, if it holds

$$\frac{1-\varphi}{\varphi}(n-1)! < 1,$$

the sum $\sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|}$ is dominated by the smallest $|R \triangle \widehat{R}|$ because $|T_o| = (n-1)!$. Then we can reduce the equation to

$$\begin{aligned} o_{\mathrm{MAP}} &= \operatorname*{argmax}_{o \in X} \sum_{\widehat{R} \in T_o \text{ s.t.} |\widehat{R} \triangle R| = \mathrm{mindiff}} \left(\frac{1-\varphi}{\varphi}\right)^{\mathrm{mindiff}} \\ &= \operatorname*{argmax}_{o \in X} \left| \left\{ \widehat{R} \in T_o \;\middle|\; |R \triangle \widehat{R}| = \mathrm{mindiff} \right\} \right|, \end{aligned}$$

where

$$\text{mindiff} = \min_{o \in X} \min_{\widehat{R} \in T_o} |R \triangle \widehat{R}|.$$

This means that we should choose the element o which is the maximum element in the most total orders which have the minimum difference with R. We call this selection based on *minimum difference* (**MND**).

Equation (8) assumes the probability for reversing the membership of a pair in R and $\widehat{R}$ to be a constant, while if we can give different two probabilities φ_+ and φ_- for the event that a pair in $\widehat{R}$ also appears in R and for the event that a pair not in $\widehat{R}$ does not appear in R, respectively, we can also give another equation as follows.

$$\begin{aligned} p(R|o) &= \sum_{\widehat{R} \in T_o} (1-\varphi_+)^{|\widehat{R}-R|} \cdot \varphi_+^{\frac{1}{2}(n^2+n)-|\widehat{R}-R|} \cdot (1-\varphi_-)^{|R-\widehat{R}|} \cdot \varphi_-^{\frac{1}{2}(n^2-n)-|R-\widehat{R}|} \\ &= \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi_+}{\varphi_+}\right)^{|\widehat{R}-R|} \cdot \varphi_+^{\frac{1}{2}(n^2+n)} \cdot \left(\frac{1-\varphi_-}{\varphi_-}\right)^{|R-\widehat{R}|} \cdot \varphi_-^{\frac{1}{2}(n^2-n)} \end{aligned} \tag{10}$$

The $\frac{1}{2}(n^2+n)$ is the number of pairs included in $\widehat{R}$, i.e. the half of all pairs $(o_1, o_2) \in X^2 (o_1 \neq o_2)$ and the n pairs of diagonal pairs. The $\frac{1}{2}(n^2-n)$ is the rest of pairs in X^2.

The probabilities φ_+ and φ_- are also estimated by the sensitivity and the specificity of induced relations, as we described in Section 3.2.

We refer to the selection based on Equation (10) as **TOA2**, which is as follows.

$$o_{\text{MAP}} = \underset{o \in X}{\text{argmax}}\, p(o) \cdot \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi_+}{\varphi_+}\right)^{|\widehat{R}-R|} \cdot \left(\frac{1-\varphi_-}{\varphi_-}\right)^{|R-\widehat{R}|}$$

4.2 First Place Assumption

The second idea concerns only the maximum object o or the first place object in an order on X, and assumes that the other objects in X have no special ranks in an ideal relation. In this case, for a pair $(o, o') \in X^2$ $(o \neq o')$ the pair is included in a guessed preference relation R with a probability, which we assume to be a constant, say ψ. A pair $(o', o) \in X^2$ is not included in the guessed R also with a probability, which is also assumed a constant probability ψ. The other pairs in X^2 should assume to be included with the probability $\frac{1}{2}$. Hence we can estimate $p(R|o)$ as follows,

$$p(R|o) = (1-\psi)^{2n-1-|A_o|} \cdot \psi^{|A_o|} \cdot \left(\frac{1}{2}\right)^{n^2-2n+1} = \left(\frac{\psi}{1-\psi}\right)^{|A_o|} \cdot (1-\psi)^{2n-1} \cdot \left(\frac{1}{2}\right)^{n^2-2n+1} \tag{11}$$

where

$$A_o = \{(o, o') \in X^2 | o \neq o' \wedge (o, o') \in R\} \cup \{(o', o) \in X^2 | o \neq o' \wedge (o', o) \notin R\}.$$

Again we have the following theorem.

Theorem 2 *When a relation R is a total order on X with the maximum element $o \in X$ and $\psi > 0.5$, then*

$$p(R|o^*) > p(R|o) \quad \textit{for all } o \in X, o \neq o^*,$$

where $p(R|o)$ is defined by Equation (11).

(See Appendix for proof) □

With Equation (11), Equation (2) is rewritten as

$$o_{\mathrm{MAP}} = \underset{o \in X}{\operatorname{argmax}}\, p(o) \cdot \left(\frac{\psi}{1-\psi}\right)^{|A_o|}. \tag{12}$$

We call a selection method by Equation (12) selection based on *the first place assumption* (**FPA**).

When we have no prior knowledge on $p(o)$ and assume

$$\frac{\psi}{1-\psi} > 1,$$

we can get

$$o_{\mathrm{MAP}} = \underset{o \in X}{\operatorname{argmax}} \left(\frac{\psi}{1-\psi}\right)^{|A_o|} = \underset{o \in X}{\operatorname{argmax}}\, |A_o|.$$

This means that we should choose the element o which has the minimum size of A_o, i.e., the set of elements in X that are less than o and elements that are not greater than o. This is the same as the winning point subtracted by the lost point in a round robin tournament. Accordingly we call this selection method selection based on *round robin tournaments* (**RRT**)

When we think of the two probabilities ψ_+ and ψ_- in the same way as φ_+ and φ_-, we have the following equation, which may be a better estimation,

$$\begin{aligned} p(R|o) &= (1-\psi_+)^{n-1-|A'_o|} \cdot \psi_+^{|A'_o|} \cdot (1-\psi_-)^{n-1-|A''_o|} \cdot \psi_-^{|A''_o|} \cdot \left(\frac{1}{2}\right)^{n^2-2n+1} \\ &= \left(\frac{\psi_+}{1-\psi_+}\right)^{|A'_o|} \cdot (1-\psi_+)^{n-1} \cdot \left(\frac{\psi_-}{1-\psi_-}\right)^{|A''_o|} \cdot (1-\psi_+)^{n-1} \cdot \left(\frac{1}{2}\right)^{n^2-2n+1}, \end{aligned} \tag{13}$$

where

$$A'_o = \{(o,o') \in X^2 | o \neq o' \wedge (o,o') \in R\} \text{ and } A''_o = \{(o',o) \in X^2 | o \neq o' \wedge (o',o) \notin R\}.$$

The probabilities ψ, ψ_+ and ψ_- are also estimated from parameter estimation examples. The selection based on Equation (13) is called **FPA2** and is formulated as

$$o_{\mathrm{MAP}} = \underset{o \in X}{\operatorname{argmax}}\, p(o) \cdot \left(\frac{\psi_+}{1-\psi_+}\right)^{|A'_o|} \cdot \left(\frac{\psi_-}{1-\psi_-}\right)^{|A''_o|}.$$

Table 1. The POS tagging problem as a selection problem.

object set	—	The set of all POS tags.
parameter space	—	Information of a word that we want to give a POS tag and its context in a sentence. This parameter space is decomposed into the word, which is the domain parameter, and other information.
domain function	—	The mapping from words to their possible POS tags. This is constituted by a lexicon.

5 Applying to Part-of-Speech Tagging

5.1 Part-of-Speech Tagging

In this section we examine the selection methods derived through Sections 3 and 4 by applying to Part-of-Speech (POS) tagging of English sentences. POS tagging is studied by ML researchers and also by ILP researchers[1,3,5,10,14]. POS tags of English words have ambiguity depending on their context where the words are used. In order to disambiguate tags a variety of approaches have been proposed. Cussens investigated an inductive concept learning method, for this problem[1,3]. He used ILP, or the ILP system Progol[16], to treat context of words in sentences and also knowledge on English grammar explicitly and in a natural way. Then competitive results to other methods were reported.

5.2 POS Tagging as a Selection Problem

The POS tagging is regarded as a selection problem, which is described in Table 1. We also used Wall Street Journal corpus[2], which includes sentences of total 3 million words tagged after the experiment of Cussens. According to his experiment we used 2/3 of words as training. The 2/3 words, i.e. 2 million words are used to induce unary or binary relations and also used to create a lexicon, which records the frequency with which words in the training set are given different tags.

The lexicon provides the domain function of our selection problem. By looking up a word in the lexicon, a set of possible tags for the word is found. What we need is to induce a relation on this set specified by the lexicon. Cussens induced the `rmv` (remove) relation, of the form,

```
rmv(FrontContext, BackContext, Tag)
```

where `FrontContext` and `BackContext` are the context or lists of tags of words before and after the target word in the sentence. Because the relation includes

[2] CD-ROM, Association for Computational Linguistics, Data Collection Initiative LDC93TI.

Table 2. Examples of for induction for POS tagging in two kinds of representation. Examples with +/− signs are positive/negative examples, respectively.

a sentence	—	Mr.Vinken is chairman of Elsevier N.V., the Dutch publishing group.
its POS tags	—	[np, np, vbz, nn, in, np, np, cma, dt, ? , vbg, nn, stp]
a target word	—	Dutch
its possible POS tags	—	np, jj, nps
context	—	[dt, cma, np, np, in, nn, vbz, np, np] and [vbg, nn, stp]
examples in unary representation	—	−rmv([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], np) +rmv([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], jj) +rmv([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], nps)
examples in binary representation	—	+prefer([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], np, jj) +prefer([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], np, nps) −prefer([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], jj, np) −prefer([dt, cma, np, np, in, nn, vbz, np, np], [vbg, nn, stp], nps, np)

one argument for a tag, we regard it as a unary relation in spite of it has three arguments. The relation eliminates possible candidates of tags. Although the `rmv` relation is the complement of the relation c_θ, it is essentially the same that we discussed. His method eliminates some tags from all of possible tags and then choose a word with the maximum frequency in the lexicon, or the prior probability $p(w)$ for a word w. This is what Equation (5) says. This is the reason that we call the selection as Cussens'. We also used induced `rmv` relations for URR, URR2 and Cussens' selection. For the other selection we used a binary relation, which has the form,

```
prefer(FrontContext, BackContext, Tag1, Tag2)
```

which compares two tags and accordingly we regard it as a binary relation. Induced `prefer` relations are used with the selection TOA, TOA2, MND, FPA, FPA2 and RRT.

Table 2 shows a sentence and examples created for both of unary and binary relations for a target word.

5.3 Tagging Procedure

We induced rules, or logic programs for the relation `rmv` and `prefer` from examples created from the 2 million words by using Progol[16]. We prepared two sets of background knowledge, one is the set that Cussens prepared and used in his experiment[1,3]. The other is a set which includes, in addition to the set prepared by Cussens, information of transition probabilities from a tag to another for each

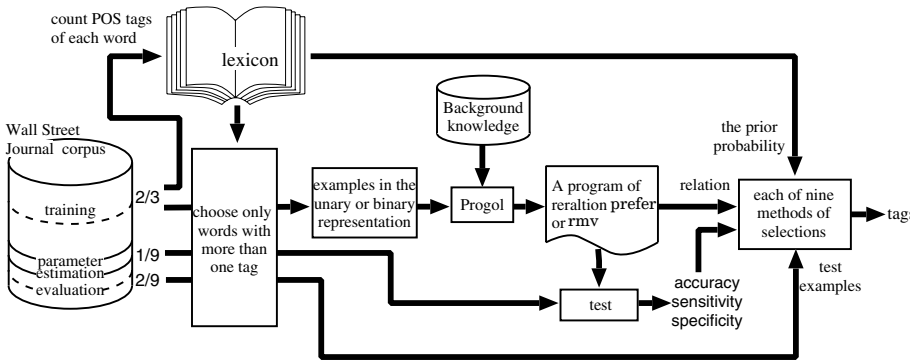


Fig. 1. Flow of the experiment.

pair of POS tags. This knowledge seems to bring effect only when we can compare two tags and is worth examining it in the binary relation representation.

The other 1/3, i.e. approximately 1 million words, are used as parameter estimation examples and as test examples. We did not use all of them for these purposes but used only 140 thousands words each of which has more than one tag candidate in the lexicon. We used 1/3 words of them as parameter estimation examples, to get the accuracy, sensitivity, and specificity of induced relations, which are necessary to estimate the maximum a posteriori tags. The other 2/3 words are for testing. Words which have ambiguity in their tags are part of them. We only used words with ambiguity for the process, the ambiguity which we can see by the lexicon. The procedure of our experiment is summarized as follows. (See also Fig.1)

1 create a lexicon from 2/3 words.
2 for each of `rmv` and `prefer`,
3 for each of two sets B and B' of background knowledge (only set B can be used for `rmv`),
4 induce a relation using Progol and a background knowledge set,
5 evaluate the accuracy, the sensitivity, and the specificity of the induced relation by words with ambiguous tags in 1/9 words, and
6 for each of selection methods (URR, URR2 and Cussens' for `rmv` and TOA, TOA2, MND, FPA, FPA2 and RRT for `prefer`),
7 adopt the selection to make tagging words with ambiguity in their tags in 2/9 words.

MND and RRT do not use the prior probability or the frequency of words in lexicon, and then they result in more than one tag for the selection. In that case we used the prior probability for selecting one.

Table 3. Accuracy, sensitivity and specificity (%) of induced relations.

			against training examples			against parameter estimation examples to acquire ρ, φ and ψ		
			accuracy	sensitivity	specificity	accuracy	sensitivity	specificity
background knowledge	B	`rmv`	85.15	75.08	98.59	80.38	68.50	95.07
		`prefer`	83.86	68.48	98.17	80.90	65.86	95.95
	B'	`prefer`	84.59	68.97	99.13	79.58	62.51	96.66

B=(the set used in [1,3]), B'=B+(transition probabilities among tags in sentences)

Table 4. Results of part-of-speech tagging using derived selection methods. Percentages (%) of words correctly tagged. Results of only words with ambiguous tags.

selection methods		unary rel.	unary relation + Bayesian		binary preference relation		binary preference relation + Bayesian			
		Cussens'	URR	URR2	MND	RRT	TOA	TOA2	FPA	FPA2
background knowledge	B	86.06	86.03	86.04	85.63	85.83	85.70	85.68	86.02	85.92
	B'	–	–	–	86.02	86.33	86.08	86.06	86.53	86.46

B=(the set used in [1,3]), B'=B+(transition probabilities among tags in sentences)

Table 5. Results of POS tagging for each case that there are n candidates. (%)

selection	Cussens'	URR	URR2	MND	RRT	TOA	TOA2	FPA	FPA2
n = 2	89.14	89.13	89.14	88.60	88.60	88.59	88.60	88.59	88.60
n = 3	76.10	76.02	76.02	76.44	76.96	76.57	76.46	77.75	77.43
n = 4	71.47	71.34	71.34	67.73	72.32	69.66	69.45	74.13	72.11

Every case uses background knowledge set used in [1,3] (B).

5.4 Results

Table 3 shows the accuracy, the sensitivity and the specificity of induced rules. We have performed three different learning, i.e., one is for `rmv` using the Cussens' background knowledge, the second is for `prefer` with the same knowledge, and the other is for `prefer` with knowledge including the knowledge of transition probabilities. The tables shows data for the training examples and for parameter estimation examples. The data for the parameter estimation examples were used as the parameters ρ, φ, ψ, etc. As we can see, each sensitivity and specificity have a large difference, and then it seems to suggest to use URR2, TOA2 and FPA2.

Table 4 summarizes the results of POS tagging, the results are the percentages of words which have been given correct tags. The selection called Cussens' traced the Cussens' experiment as much as possible. The proposed methods gave competitive results to the Cussens' but Cussens' had still the best result within the selection methods using the background knowledge denoted by B.

The knowledge including transition probability raises the results in all selection methods. FPA had the highest result with background knowledge B' and is higher than the Cussens'.

We do not intend to conclude that FPA is the best method, because the best result was produced with the additional knowledge. Table 5 shows the percentages of correctly tagged cases by the sizes of sets of candidate tags. We can see that binary methods have better results than the Cussens' or other unary method in the case of $n = 2$. On the other hand, binary methods, especially FPA, have better results in $n = 3$ or 4. In the case of $n = 4$ FPA has 2.5% higher than unary methods.

6 Conclusions

We dealt with selection learning within the framework of concept learning and Bayesian learning. Transformation of selection learning to concept learning were investigated. One is to represent selection using a unary relation on objects to be selected and the other is using a binary relation on objects. The ILP learning is applied to induce the relations.

We derived the selection of objects based on Bayes theorem in order to estimate maximum posteriori selection by using induced relations, under the several assumptions and conditions. The method that Cussens used in his tagging experiment and also methods using the minimum difference or round robin tournaments were explained in the framework. Table 6 summarize the selection methods derived.

In the derivation we assumed constant error of each elements of relations. Although URR2, TOA2 and FPA2 refined this assumption, they did not necessarily refine the results. A possible reason of this is that the assumption is too strong. Estimation of probability by analyzing induced rules by probability theory is necessary to give accurate probability.

TOA assumes the total order and estimates are based on the distance from them, and accordingly it gave smaller probability to relations which is far from a total order. FPA is based only the maximum object in an order. This seems a reason that the FPA had better results, although we need more detailed analysis.

In the Cussens' experiment, POS tags that appear with the frequency less than 5% for each word are deleted from lexicon. This process has effect of cleaning the lexicon data. Although our experiments also used this process, it is possible that we would have better results without it, because we more appropriately estimate probability. The other possibility to improve our results includes rearrangement of learning parameters, such as the number of examples and the depth of search, because the binary representation has larger hypothesis space.

Acknowledgments. We would like to thank Dr. James Cussens for giving us information about his experiment and providing us background knowledge set for the experiment. Thanks also to anonymous referees for useful comments.

Table 6. Selection methods derived and examined.

selection	representation	assumption	conditions	maximum a posteriori objects
URR	unary	const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \left(\frac{1-\rho}{\rho}\right)^{\lvert R \triangle \{o\}\rvert}$
URR2	unary	directional const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \left(\frac{1-\rho_+}{\rho_+}\right)^{\lvert \{o\}-R\rvert} \cdot \left(\frac{1-\rho_-}{\rho_-}\right)^{\lvert R-\{o\}\rvert}$
Cussens'	unary	const. error	$(\frac{1-\rho}{\rho})^2 \max_{o \in X} p(o) < \min_{o \in X} p(o)$	$\underset{o \in X \cap R}{\text{argmax}}\ p(o)$ if $R \neq \emptyset$ or $\underset{o \in X}{\text{argmax}}\ p(o)$ otherwise
TOA	binary	total order, const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{\lvert R \triangle \widehat{R}\rvert}$
TOA2	binary	total order, directional const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi_+}{\varphi_+}\right)^{\lvert \widehat{R}-R\rvert} \cdot \left(\frac{1-\varphi_-}{\varphi_-}\right)^{\lvert R-\widehat{R}\rvert}$
MND	binary	total order, const. error	$\frac{1-\varphi}{\varphi}(n-1)! < 1$ no prior knlg.	$\underset{o \in X}{\text{argmax}}\ \left\lvert \left\{ \widehat{R} \in T_o \right\rvert\ \lvert R \triangle \widehat{R}\rvert = \text{mindiff} \right\rvert$, where $\text{mindiff} = \min_{o \in X} \min_{\widehat{R} \in T_o} \lvert R \triangle \widehat{R}\rvert$
FPA	binary	first place, const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \left(\frac{\psi}{1-\psi}\right)^{\lvert A_o\rvert}$, where $A_o = \{(o,o') \in X^2 \mid o \neq o' \wedge (o,o') \in R\} \cup \{(o',o) \in X^2 \mid o \neq o' \wedge (o',o) \notin R\}$
FPA2	binary	first place, directional const. error	–	$\underset{o \in X}{\text{argmax}}\ p(o) \cdot \left(\frac{\psi_+}{1-\psi_+}\right)^{\lvert A'_o\rvert} \cdot \left(\frac{\psi_-}{1-\psi_-}\right)^{\lvert A''_o\rvert}$, where $A'_o = \{(o,o') \in X^2 \mid o \neq o' \wedge (o,o') \in R\}$ $A''_o = \{(o',o) \in X^2 \mid o \neq o' \wedge (o',o) \notin R\}$
RRT	binary	first place, const. error	$\frac{\psi}{1-\psi} > 1$ no prior knlg.	$\underset{o \in X}{\text{argmax}}\ \lvert A_o\rvert$, where $A_o = \{(o,o') \in X^2 \mid o \neq o' \wedge (o,o') \in R\} \cup \{(o',o) \in X^2 \mid o \neq o' \wedge (o',o) \notin R\}$

References

1. J. Cussens. "Part-of-Speech Disambiguation Using ILP", *Technical Report PRG-TR-25-96*, Oxford University Computing Laboratory, 1996.
2. J. Cussens. "Bayesian Inductive Logic Programming with Explicit Probabilistic Bias", *Technical Report PRG-TR-24-96*, Oxford University Computing Laboratory, 1996.
3. J. Cussens. "Part-of-Speech Tagging Using Progol", *Proc. 7th Int'l Workshop on ILP, LNAI 1297*, pp. 93–108, Springer, 1997
4. J. Cussens. "Using Prior Probabilities and Density Estimation for Relational Classification", *Proc. 8th Int'l Conf. on ILP, LNAI 1446*, Springer, pp. 106–115, 1998.

5. J. Cussens, S. Džeroski and T. Erjavec. "Morphosintactic Tagging of Solovene Using Progol", *Proc. 9th Int'l Workshop on ILP, LNAI 1634*, Springer, pp. 68–79, 1999.
6. H. A. David. "Ranking from Unbalanced Paired-Comparison Data", *Biometrika* **74**, pp. 432–436, 1987
7. L. Dehaspe. "Maximum Entropy Modeling with Clausal Constraints", *Proc. 7th Int'l Workshop on ILP, LNAI 1297*, Springer, pp. 109–124, 1997.
8. P. Flach and N. Lachiche. "1BC: A First Order Bayesian Classifier", *Proc. 9th ILP, LNAI 1634*, Springer, pp. 92–103, 1999.
9. J. D. Hirst, R. D. King and M. J. E. Sternberg, "Quantitative structure-activity relationships by neural networks and inductive logic programming. I. The inhibition of dihydrofolate reductase by pyrimidines", *Journal of Computer-Aided Molecular Design 8 (1994)*, pp. 405-420, 1994.
10. T. Hotváth, Z. Alexin, T. Gyimóthy and S. Wrobel. "Application of Different Learning Methods to Hungarian Part-of-Speech Tagging", *Proc. 9th Int'l Workshop on ILP, LNAI 1634*, Springer, pp. 128–139, 1999.
11. N. Inuzuka, T. Nakano and H. Itoh. "Acquiring Heuristic Value in Search Trees by a Relational Learner", *Proc. 11th Int'l Conf. on Applications of Prolog*, pp. 157-162, 1998.
12. J Jelonek and J Stefanowski. "Experiments on Solving Multiclass Learning Problems by n^2-classifier", *Proc. 10th European Conf. on Machine Learning, LNAI 1398*, Springer, pp. 172–177, 1998.
13. D. Koller. "Probabilistic Relational Models". *Proc. 9th ILP, LNAI 1634*, Springer, pp. 3–13, 1999.
14. L. Márquetz and H. Rodríguez. "Part-of-Speech Tagging Using Decision Trees", *Proc. 10th European Conf. on Machine Learning, LNAI 1398*, Springer, pp. 25–36, 1998.
15. S. Muggleton. "A Learning Model for Universal Representations", *Proc. 4th Int'l Workshop on ILP*, pp. 139–160, 1994.
16. S. Muggleton. "Inverse Entailment and Progol", *New Generation Computing Journal*, **13** pp. 245–286, 1995.
17. T. Nakano, N. Inuzuka, H. Seki and H. Itoh. "Inducing Shogi Heuristics Using Inductive Logic Programming", *Proc. 8th Int'l Conf. on ILP, LNAI 1446*, Springer, pp. 155–164. 1998.
18. S. Slattery and M Craven. "Combining Statistical and Relational Methods for Learning in Hypertext Domains", *Proc. 8th Int'l Conf. on ILP, LNAI 1446*, Springer, pp. 38–52, 1998.

A Proofs

Proof of Theorem 1 First of all we think of a bijection from T_{o^*} to T_o denoted by g. The bijection g maps a total order R_1, which means

$$o^* > o_{i_1} > \cdots > o_{i_k} > o > o_{i_{k+1}} > \cdots,$$

in T_o^* to a total order $g(R_1) = R_2$, which means

$$o > o_{i_1} > \cdots > o_{i_k} > o^* > o_{i_{k+1}} > \cdots,$$

and is in T_o. R_2 is only different form R_1 in the order of o^* and o. Then,

$$\begin{aligned} R_1 - R_2 &= \{(o^*, o_{i_1}), \cdots, (o^*, o_{i_k}), (o^*, o), (o_{i_1}, o), \cdots, (o_{i_k}, o)\}, \\ R_2 - R_1 &= \{(o, o_{i_1}), \cdots, (o, o_{i_k}), (o, o^*), (o_{i_1}, o^*), \cdots, (o_{i_k}, o^*)\}. \end{aligned}$$

Reminding that the maximum in R is o^*, the first $k+1$ pairs of $R_1 - R_2$ agree with R and the last $k+1$ pairs of $R_2 - R_1$ disagree with R. Hence, we have

$$|(R_1 - R_2) \cap R| \geq k+1 \quad \text{and} \quad |(R_2 - R_1) - R| \geq k+1,$$

and also it holds

$$|(R_1 - R_2) - R| \leq k \quad \text{and} \quad |(R_2 - R_1) \cap R| \leq k.$$

Using these equations and also $|A| = |A \cup B| + |A - B|$, $(A - B) - C = (A - C) - B$, $(A - B) \cap C = (C - B) \cap A$, we can calculate as follows.

$$\begin{aligned} |R \triangle R_1| &= |R - R_1| + |R_1 - R| \\ &= |(R - R_1) \cap R_2| + |(R - R_1) - R_2| + |(R_1 - R) \cap R_2| + |(R_1 - R) - R_2| \\ &= |(R_2 - R_1) \cap R| + |(R - R_1) - R_2| + |(R_2 - R) \cap R_1| + |(R_1 - R_2) - R| \\ &\leq k + |(R - R_1) - R_2| + |(R_2 - R) \cap R_1| + k \\ |R \triangle R_2| &= |R - R_2| + |R_2 - R| \\ &= |(R - R_2) \cap R_1| + |(R - R_2) - R_1| + |(R_2 - R) \cap R_1| + |(R_2 - R) - R_1| \\ &= |(R_1 - R_2) \cap R| + |(R - R_1) - R_2| + |(R_2 - R) \cap R_1| + |(R_2 - R_1) - R| \\ &\geq (k+1) + |(R - R_1) - R_2| + |(R_2 - R) \cap R_1| + (k+1) \end{aligned}$$

Hence we have $|R_1 \triangle R| < |g(R_1) \triangle R|$. As a result if $\varphi > 0.5$ it holds

$$p(R|o^*) = \sum_{\widehat{R} \in T_{o^*}} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|} \cdot \varphi^{n^2} > \sum_{\widehat{R} \in T_{o^*}} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle g(\widehat{R})|} \cdot \varphi^{n^2} = \sum_{\widehat{R} \in T_o} \left(\frac{1-\varphi}{\varphi}\right)^{|R \triangle \widehat{R}|} \cdot \varphi^{n^2} = p(R|o)$$

□

Proof of Theorem 2 If it holds $|A_{o^*}| > |A_o|$ for every $o \in X, o^* \neq o$ then we can have the theorem. Obviously $|A_{o^*}|, |A_o| \leq 2(n-1)$. It is also obvious that $|A_{o^*}| = 2(n-1)$ because R is a total order with the maximum element o^*. On the other hand, for $o \neq o^*$, $(o, o^*) \notin A_o$ because $(o, o^*) \notin R$, and $(o^*, o) \notin A_o$ because $(o^*, o) \in R$. Hence $|A_o| \leq 2(n-1) - 2$ and then $|A_{o^*}| > |A_o|$. □

Concurrent Execution of Optimal Hypothesis Search for Inverse Entailment

Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi

Faculty of Sci. and Tech.,
Science University of Tokyo
Noda, Chiba, 278-8510, Japan
{ohwada, nisiyama, mizo}@imc.sut.ac.jp

Abstract. Inductive Logic Programming (ILP) allows first-order learning and provides greater expressiveness than propositional learning. However, due to its tradeoff, the learning speed may not be reasonable for datamining settings. To overcome this problem, this paper describes a distributed implementation of an ILP engine, allowing speeding up optimal hypothesis search in inverse entailment according to the number of processors. In this implementation, load balancing is achieved by contract net communication between the processors, resulting in a dynamic allocation of the hypothesis search task. This paper describes our concurrent search algorithm, distributed implementation and experimental results for speeding up inverse entailment. An initial experiment was conducted to demonstrate the well-balanced task allocation.

1 Introduction

While Inductive Logic Programming (ILP) has greater expressive power as a relational learning framework than do propositional learners, the learning speed of ILP is relatively slower due to the tradeoff between expressive power and efficiency. A typical ILP system, Progol, formalizes induction as a hypothesis space search problem to obtain optimal hypotheses with maximum compression, and provides efficient implementation of the optimal search [Muggleton95]. However, the number of hypotheses generated during the search is still very large; there may be no effective method for constraining a general hypothesis search. New implementation schemes from computer science technology are worth considering to realize efficient ILP systems.

A large amount of data must be processed efficiently for datamining applications of ILP. Recent attempts to resolve this problem have focused on a sampling method based on statistics [Srinivasan99] and on information decomposition by checking the independency among the examples and background knowledge [Blockeel99]. While these attempts have focused on large data sets, we will focus on a large hypothesis space in which a parallel hypothesis search could be exploited. A first attempt to design a parallel ILP system can be found in [Fujita96]. In this first attempt, a first-order theorem prover is used as an interpreter on a concurrent logic programming language. However, performance

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 165–173, 2000.

issues were not reported. A parallel version of FOIL has been attempted in [Matui98], but scalability could not be obtained due to the number of parallel processors.

In our previous study [Ohwada99], we attempted concurrent execution of an ILP system based on the framework of concurrent logic programming to overcome the ILP efficiency problem. We implemented a concurrent ILP engine using a concurrent logic programming language, KL1 (developed in the Japanese Fifth Generation Computer Project). We subsequently demonstrated that the experimental result of a benchmark test set coincided with the expected degree of parallelism, using a shared-memory machine with six processors. However, KL1 is a committed choice language, so background knowledge could not be expressed directly by a clausal form.

In this research, we propose a combination of logic programming and concurrent logic programming where logic programming is used to solve goals and concurrent logic programming dispatches the goals to distributed machines. Here, a goal means a hypothesis search task, and is independently solved within each machine. In this implementation, load balancing is achieved by contract net communication between the machines, resulting in a dynamic allocation of the hypothesis search task. This paper also describes our concurrent search algorithm, distributed implementation and experimental results for speeding up an optimal hypothesis search in inverse entailment.

2 Hypothesis Search in ILP

Why is parallel hypothesis search needed? In order to realize an efficient search in ILP, heuristics such as information gain and pruning information are important. From a knowledge discovery point of view, it is desirable to avoid reducing the hypothesis space. As a result, the hypothesis space becomes larger. However, most engineering tools support concurrent execution of the underlying computation. In this sense, we adopt a parallel hypothesis search to realize efficient ILP systems.

We provide here concurrent execution of hypothesis search in inverse entailment employed within Progol. Our ILP system mentioned in this paper works in the same way as Progol does under a single processor machine.

To explain a hypothesis space search problem simply, we suppose that a set of positive examples (E^+), a set of negative examples (E^-), and background knowledge(B) are collections of ground terms[1]. An ILP system sequentially produces h_i to satisfy the following definite clauses:

$$h_1 \wedge ... \wedge h_n \wedge B \models E^+$$
$$h_1 \wedge ... \wedge h_n \wedge B \wedge E^- \not\models \square$$

where h_i is called a hypothesis in the following.

[1] This restriction is just for explanation.

Progol constructs a bound hypothesis space using the input-output modes of predicate arguments and the depth of variable connectivity. Such a space forms a lattice, the bottom of which is the most specific clause, which explains the example $e \in E^+$. For each hypothesis h_i and h_j in the lattice, h_i is an upper-level hypothesis of h_j, if a substitution θ exists such that $h_i\theta \subseteq h_j$. In this case, h_j is a more specific clause of h_i. We denote $Refine(h)$ as the set of more specific clauses of h.

Progol finds a hypothesis from this space. A preferred hypothesis covers as many positive examples as possible and as few negative examples as possible. Let $p(h)$ and $n(h)$ be the numbers of positive and negative examples covered by hypothesis h. The number of literals in h is denoted by $c(h)$. We express them as

$$g(h) = p(h) - c(h),$$
$$f(h) = g(h) - n(h)$$

where $g(h)$ indicates the generality of h and $f(h)$ indicates the compression measure, as used in Progol. $g(h)$ monotonically decreases for a top-down hypothesis search, while $f(h)$ is non-monotonic. Progol finds hypothesis h with maximum compression in which $n(h) = 0$, and there is no hypothesis h' such that $f(h) \geq g(h')$. However, noise must be considered. Suppose that a small number of negative examples (n^*) is allowed for noise. Even if Progol finds a hypothesis (h) that satisfies $n(h) \leq n^*$, Progol must search for the optimal hypothesis when hypotheses more specific than h must be explored.

Such a hypothesis search is formulated in the following optimization problem:

$$\texttt{maximize}\ f(h)\ \texttt{subject to}\ n(h) \leq n^*$$

This problem can be solved by a branch-and bound search that finds an optimal hypothesis. There are two parameters, *Active* and *Incumbent*, for the search algorithm. *Active* is a set of nodes that are not terminal but are frontier during the search. *Incumbent* is a terminal node but is a candidate solution such that $n(h) \leq n^*$. The initial value of *Active* is a singleton whose element is a most general hypothesis. The initial value of *Incumbent* is null. The search algorithm is as follows:

```
Branch(Active, Incumbent)
1 while Active ≠ ∅
2   do select h from Active using f(h);
3      Active ← Active − {h};
4      for each c ∈ Refine(h);
5          do Bound(c);
6 return Incumbent;
```

The algorithm selects a hypothesis (h) using $f(h)$ until *Active* becomes empty, and refines the hypothesis by the function $Refine$. The procedure *Bound* determines whether a hypothesis h should be further refined or not. This procedure calculates the values of $f(h)$ and $n(h)$. The current incumbent might be

updated if $n(h) \leq n^*$. Otherwise, h is put into *Active*. Thus, the procedure can be defined as follows:

```
Bound(c)
7   if Neg(c) ≤ n*
8       then {updating the incumbent}
9           if Neg(Incumbent) > Neg(c)
10              then Incumbent ← c;
11      else
12          Active ← Active ∪ {c};
```

Although the working program implements a more complex algorithm, this simple program is sufficient to explain the parallel algorithm of a hypothesis search. This algorithm is a branch-and-bound search version of Progol.

3 Parallel Hypothesis Search

A parallel hypothesis search explores clauses included in *Active* in parallel. We divide *Active* into two sets $Left$ and *Right* where *Right* is explored within this process and $Left$ is within another process. This parallel version of the Branch procedure in Section 2 is designed as follows:

```
Parallel-Branch(Active, Incumbent)
13 while Active ≠ ∅
14   do if a new candidate solution (d) is
15           received from another process,
16      then if Neg(Incumbent) > Neg(d)
17              then Incumbent ← d;
18           else if there is no requesting task
19              then Active' = Left(Active);
20                   Request&Commit(Active', Incumbent);
21                   Active = Right(Active);
22      select h from Active;
23      Active ← Active − {h};
24      for each c ∈ Refine(h);
25          do Bound(c);
26 return Incumbent;
```

where the functions $Left$ and *Right* divide *Active* into two subsets. The Parallel-Branch procedure consists of the following three cases. In the first case, another process sends a new candidate solution(d). If d is better than the incumbent, the incumbent is updated in Line 16-17. In the second case, there is no other process being requested to perform a hypothesis search. In this case, *Active* is divided. The set obtained by $Left$ is sent to other processes, and the processes are requested to search for the set in Lines 19-20. The set obtained by *Right*

is added to *Active*, and solved within this Parallel-Branch process. In the third case, there is a process to request a hypothesis search, in which case, Lines 16 to 19 are executed. Figure 1 illustrates the flow of the algorithm.

The Parallel-Branch algorithm also works in the processes requested for hypothesis search. *Active* is also decomposed into a smaller set that is sent to other processes.

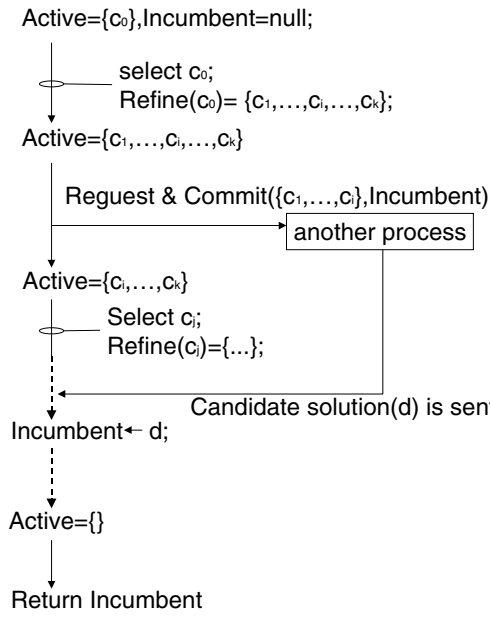


Fig. 1. Flow of parallel hypothesis search

A requested process for hypothesis search is determined by the reply from other processes at run time. For this purpose, we adopt a contract net protocol used in the multi-agent community[Smith 80]. We simplify this protocol in that a requested process is determined by the three messages, *request*, *accept* and *commit*. A request message for hypothesis search is broadcasted to all the processes that may reply with accept messages. The original process selects one process from the processes that send the accept messages, and sends a commit message to the process, as shown in Figure 2. In this figure, two of the three task receivers are idle and reply with accept messages. The task sender selects one of the two processes, and commits the hypothesis search to the selected process by sending a commit message.

After the hypothesis search is committed to a process, a new candidate solution may be sent from the process, and *incumbent* may be updated. When the original process finishes the hypothesis search, the obtained incumbent becomes the best hypothesis whose $f(h)$ is optimal.

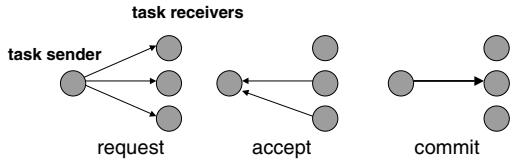


Fig. 2. Contact net communication

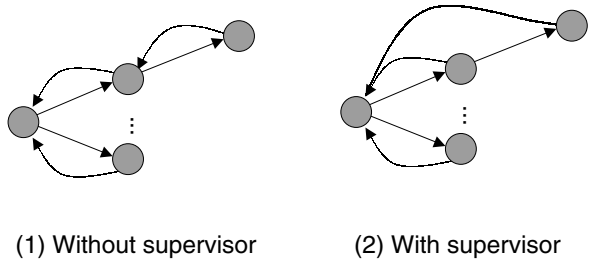


Fig. 3. Two types of interprocess communication

We employ two types to receive new candidate solutions. The first type waits to receive a candidate solution from another process that executes the hypothesis search. The another type provides a supervisor that receive every candidate solution set from all the processes. These types are shown in Figure 3. The leftmost process (depicted as a circle) is the initial process of a hypothesis search; in general this process requests several processes to perform the search. Message passing of the first type is done in a hierarchical manner. In contrast, the second type constructs a flat structure in which all hypothesis search processes send candidate solutions to the initial process.

4 Implementation

The Parallel-Branch procedure in the learning algorithm is implemented in standard logic programming systems (*e.g.*, Eclipse and Gnu-prolog), and the Request&Commit procedure is implemented in a concurrent logic programming language, KL1. KLIC[Chikayama97] compiles a KL1 program into the corresponding C program that is linked with the Parallel-Branch program. These programs are available by anonymous ftp from the site (http://www.ia.noda.sut.ac.jp/ilp).

Progol maintains triplet $<C,\theta,k>$ for hypothesis searches. In this triplet, C is a clause, θ is a substitution and k is an index. In order to request other processes to perform the hypothesis search, the Parallel-Branch procedure sends a bottom clause and the current incumbent as well as *Active* whose element is a triplet. Note that previously explored clauses (such as a close list) may not be sent because the hypothesis search space constructs a tree (no loop).

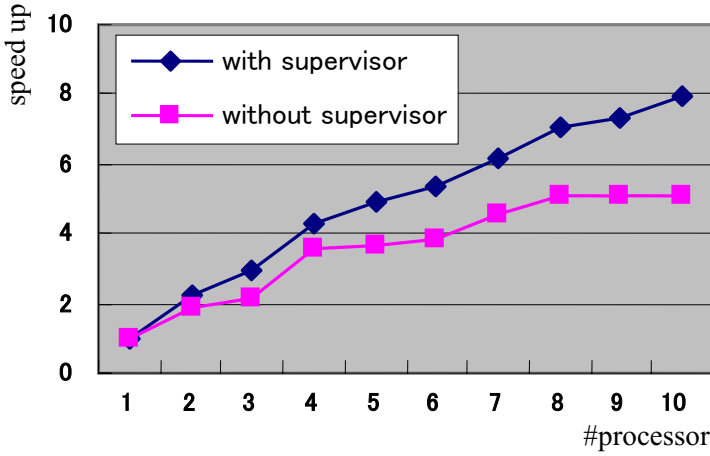


Fig. 4. Speed up ratio for the number of processors.

5 Experiment

To show the scalability of the concurrent execution of a hypothesis search, we measured the execution time of our ILP system for ten-processor parallel machines (Sun Enterprise 5000). Data sets are the KRK-illegality of a chess endgame problem[Muggleton 89], and our business application of sales promotion where rules for good customers are produced from relationships among sales item, price, customer characteristics, weather, and so on. The first problem includes 342 positive and 658 negative examples, and our ILP system generates the same rules as Progol does. The latter problem includes 253 positive and 8979 negative examples. Background knowledge consists of 20 predicates, and the total size of the data set is 4078KB. The data set is also available by anonymous ftp. Our ILP system for example produces the following rule:

```
good_custommer(A) :-
    ordered(A,ID),
    order_time(ID,Date,Time),
    total_price(ID,Price),
    low_price(Price),
    saturday(Data),
    hot(Data,Time).
```

The proposed algorithm works on one to ten processors. The first experiment measured the learning times to compare the two types of interprocess communication mentioned in Section 3. We selected the sales promotion data set only, because KRK-illegality could be learned in a short time. Table 1 shows the result. Learning time on a single processor was 502 second, and the speed up ratio increased according to the number of processors. As shown in the graph,

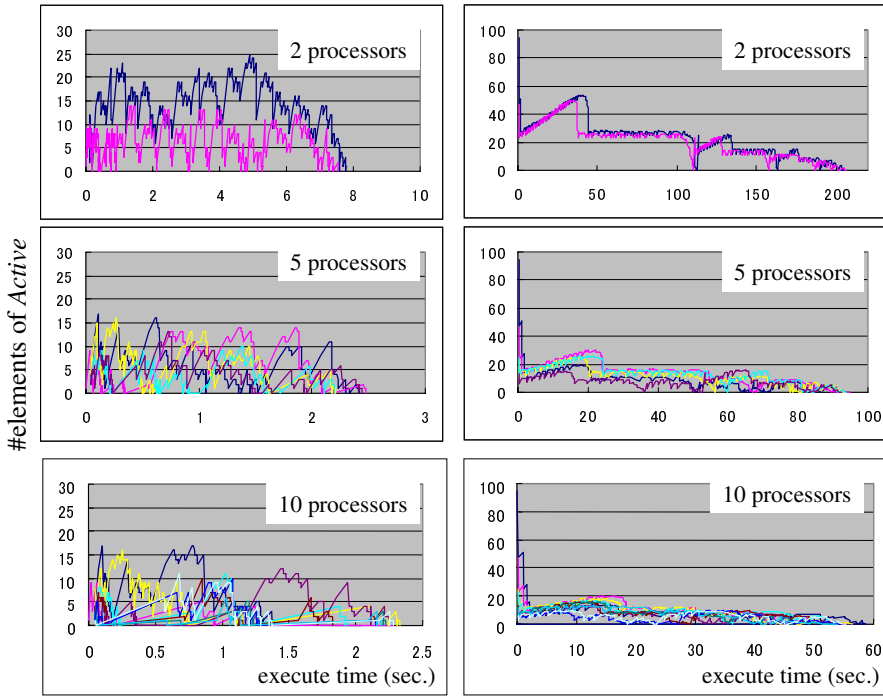


Fig. 5. Search execution times using from 2 to 10 processors.

the supervisor-based approach has high parallelism because the waiting time to receive the candidate solution could not be ignored in the hierarchical approach. In contrast, the supervisor-based approach is centralized processing, but its computation cost is relatively small when only the incumbent is updated.

The second experiment was conducted to investigate how well distributed elements of *Active* are. Figure 5 shows the result of learning for KRK-illegality and sales promotion using from two to ten processors. For KRK-illegality, *Active* is small, and thus it is hard to distribute elements of *Active* uniformly. In contrast, search tasks are uniformly distributed for sales promotion. This indicates that scalability is obtained in a parallel hypothesis search when a given ILP problem is large.

6 Conclusions

This paper described a method for parallelizing an ILP engine based on inverse entailment and implementing the engine on a parallel machine environment. A hypothesis search is performed independently within each, and a set of nodes to

be explored can be decomposed into two sets. The search task is dynamically allocated to each machine using a contract net communication. The experimental result shows that task allocation is well-balanced and scalable for large ILP problems.

References

[Blockeel99] Blockeel, H., De Raedt, L., Jacobs, N. and Demoen, B., Scaling Up Inductive Logic Programming by Learning from Interpretations, *Data Mining and Knowledge Discovery*, Vol. 3, No. 1, pp. 59–94, 1999.

[Chikayama97] Chikayama, T., *KLIC User's Manual*, Institute for New Generation Computer Technology, 1997.

[Fujita96] Fujita, H., Yagi, N., Ozaki, T., and Furukawa, K., A new design and implementation of Progol by bottom-up computation, *Proc. of the 6th International Workshop on ILP*, pp. 163–174, 1996.

[Muggleton89] Muggleton, S. H., Bain, M. E., Michie, D., An experimental comparison of human and machine learning formalism. *Proc. of the Sixth International Workshop on Machine Learning*, 1989.

[Matui98] Matsui, T., Inuzuka, N., Seki, H. and Itoh, H., Parallel Induction Algorithms for Large Samples, *Proc. of the First International Conference on Discovery Science*, pp. 397–398, 1998.

[Muggleton95] Muggleton, S., Inverse Entailment and Progol, *New Generation Computing*, Vol. 13, Nos. 3,4, pp. 245–286, 1995.

[Ohwada99] Ohwada, H. and Mizoguchi, F., Parallel Execution for Speeding Up Inductive Logic Programming Systems, *Proc. of the Second International Conference on Discovery Science*, pp. 277-286, 1999.

[Smith80] Smith, R., The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver, *IEEE Transactions on Computers*, Vol. C-29, No.12, 1980.

[Srinivasan99] Srinivasan, A., A study of Two Sampling Methods for Analyzing Large Datasets with ILP, *Data Mining and Knowledge Discovery*, Vol. 3, No. 1, pp. 95–123, 1999.

Using ILP to Improve Planning in Hierarchical Reinforcement Learning

Mark Reid and Malcolm Ryan

School of Computer Science and Engineering, University of New South Wales
Sydney 2052, Australia
{mreid,malcolmr}@cse.unsw.edu.au

Abstract. Hierarchical reinforcement learning has been proposed as a solution to the problem of scaling up reinforcement learning. The RL-TOPs Hierarchical Reinforcement Learning System is an implementation of this proposal which structures an agent's sensors and actions into various levels of representation and control. Disparity between levels of representation means actions can be misused by the planning algorithm in the system. This paper reports on how ILP was used to bridge these representation gaps and shows empirically how this improved the system's performance. Also discussed are some of the problems encountered when using an ILP system in what is inherently a noisy and incremental domain.

1 Introduction

Reinforcement learning [15] has been studied for many years now as approach to learning control models for robot or software agents. It works superbly on small, low-dimensional domains but has trouble scaling up to larger, more complex, problems. These larger problems have correspondingly larger state spaces which means random exploration and reward back-propagation – the key techniques in reinforcement learning – are much less effective.

Hierarchical reinforcement learning has been proposed as a solution to this lack of scalability and comes in several forms ([12], [16], [11], [3]). The overarching idea is to break up a large reinforcement learning task into smaller subtasks, find policies for the subtasks, then finally recombine them into a solution for the original, larger problem. The approach examined in this paper, the RL-TOPs Hierarchical Reinforcement Learning System[13], represents the agent's state symbolically at various levels of abstraction. This allows for a unique synthesis of reinforcement learning and symbolic planning.

Low-level state representation is ideal when an agent requires a fine-grained view of its world (eg, motor control in robot walking and balancing). In order to make larger task tractable, however, a higher level of abstraction is sometimes required (eg, moving the robot through several rooms in an office block). In the RL-TOPs system the high-level representation of a problem is provided by a domain expert who defines coarse-grained actions and states for the agent. The

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 174–190, 2000.

low-level implementation of the coarse actions are left to the agent to invent using reinforcement learning.

The high-level states and actions do not always convey every relevant feature of an agent's state space to the planning side of the RL-TOP system. The work presented in this paper shows that by examining the interplay between the agent's low-level and high-level actions, new high-level features can be constructed using a standard ILP algorithm.

The paper is organized as follows. Section 2 gives an overview of the RL-TOP system and the planning problem motivating the use of ILP to learn new state space features. Section 3 details the transformation of the planning problem into an ILP learning task as well as outlining a method for converting a batch learning algorithm into an incremental learner. Finally, Section 4 describes a simple domain used to test the performance of the RL-TOPs system augmented with an ILP system. An experiment comparing the performance of three reinforcement learning approaches on this domain is then analyzed.

2 Reinforcement-Learnt Teleo-Operators

The key concept in the hierarchical reinforcement system presented in this paper is the *Reinforcement-Learnt Teleo-OPerator* or RL-TOP [12]. It is a synthesis of ideas from planning and reinforcement learning. Like Nilsson's teleo-operators (TOPs) [10], RL-TOPs define agent behaviours by symbolically describing their preconditions and effects. The main advantage RL-TOPs have over standard TOPs is that the agent does not need hard-coded instructions on how to carry out each of its behaviours. Instead, the problem of moving from the set of states defined by an RL-TOP's precondition to the states defined by its intended effect is treated as a reinforcement learning task. This allows a large reinforcement learning task to be broken down into several easier ones which are combined together by a symbolic planner.

The use of RL-TOPs in the system presented in this paper is analogous to the use of *options* as described by Sutton et al in [15], the *Q nodes* of Dietterich's MAXQ system [3] as well as *abstract actions* and *macro-actions* described elsewhere in the literature. The RL-TOP approach distinguishes itself from these other systems by emphasizing the symbolic representation of agent behaviours.

2.1 Definitions and Notation

At any point in time an agent is assumed to be in some *state* $s \in \mathcal{S}$. To move from one state to another an agent has a finite number of *actions* $\mathcal{A} = \{a_1, \ldots, a_k\}$ which are functions from $\mathcal{S}$ to $\mathcal{S}$. Actions need not be deterministic.

Given a set of *goal states* $\mathtt{G} \subset \mathcal{S}$ a reinforcement learning task requires an agent to come up with a *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$. A *good* policy is one that can take an agent from any initial state, $s_0 \in \mathcal{S}$, through a sequence of steps, $s_0, \ldots, s_n$, such that $s_n \in \mathtt{G}$. A *step* is made from s_i to s_{i+1} by applying the action $\pi(s_i)$

to the state s_i yielding s_{i+1}. An *optimal* policy is one that for any initial state generates the shortest possible sequence of steps to a goal state.

An RL-TOP or *behaviour* B consists of three components, a *precondition*, a *policy*, and a *postcondition* (or effect). The precondition, denoted B.pre, is a set of states in which the behaviour is applicable. The postcondition, B.post, is another set of states which define a behaviour's intended effect. An agent's behaviour is executed by using its policy, $\mathsf{B}.\pi$, to move from state to state until the agent is no longer in that behaviour's precondition. If, when the execution of the behaviour terminates, the agent is in B.post the behaviour was said to be *successful*. Behaviours are given to the agent by defining its pre- and post-conditions. Finding good policies for each behaviour then becomes a standard reinforcement learning task – steps that lead to a successful execution of a behaviour are rewarded while steps that leave B.pre without ending up in B.post are punished. The implementation details of this reinforcement learning is not sufficiently pertinent to warrant discussion in this paper. We point the interested reader to [13] and [12].

A finite set of predicates, $\mathcal{P} = \{\mathtt{P}_1, \ldots, \mathtt{P}_m\}$, is called *primitive* if every state $s \in \mathcal{S}$ can be uniquely identified with a conjunction of ground instances of these predicates. A primitive set of predicates allows subsets of the state space to be described by new predicates defined in terms of disjunctions of conjunctions built from elements of $\mathcal{P}$. We call these non-primitive predicates *high-level* predicates. In the remainder of this discussion it will be assumed that goals, preconditions, postconditions and other subsets of $\mathcal{S}$ are defined by high-level predicates.

2.2 Planning and the Frame Axiom

The symbolic representation of behaviours' preconditions and postconditions provides the link between planning and reinforcement learning. As sets they define small reinforcement learning tasks and symbolically it becomes possible to build *Teleo-Reactive (TR) plan trees*. We will briefly explain their construction and limitations through a few examples. A more detailed exposition can be found in [2].

Suppose an agent has a goal G and four behaviours: A, B, C, and D. The left-hand side of Figure 1 is an abstract representation of the agent's state space showing the set of goal states as a shaded rectangle and behaviours as labelled arrows connecting their pre- and postcondition sets.

Since, as the figure shows, that A.post $\subset$ G, it must be the case if the agent is in A.pre and successfully executes behaviour A it will achieve the goal G. It is important to note that this subset test is actually performed in the planning algorithm by testing if the high-level predicate for G subsumes that for A.pre. The problem of reaching a state in G has now been reduced to getting to a state in G *or* getting to a state in A.pre and executing behaviour A. This process is repeated with A.pre as the new goal and B.pre $\cup$ C.pre $\cup$ D.pre are then found to be states from which it is possible to achieve the goal G. The TR tree on the right of the figure shows which actions the agent needs to successfully execute to reach the goal. For example, if the agent state is in the set defined by D.pre it

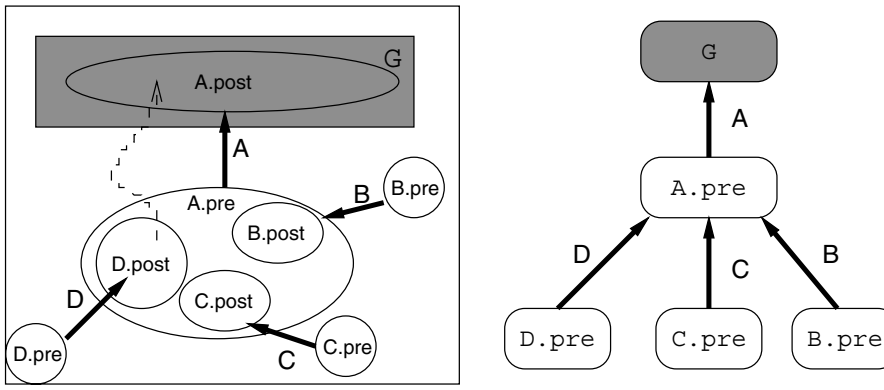


Fig. 1. The diagram on the left shows four behaviours and a goal in a state space. On the right is the corresponding TR plan tree. The dashed line represents a sequence of steps which successfully execute behaviour A

needs to execute behaviour D followed by behaviour A. If an agent moves from a node of a TR tree into a node which is not the one that was expected a *plan failure* is said to have occurred.

The above process for generating TR trees works if the goal state completely subsumes some behaviour's postcondition. The goal in Figure 2, however, is a conjunction of two predicates, $\mathtt{G}_1 \wedge \mathtt{G}_2$, and only one of them, $\mathtt{G}_1$, subsumes A's postcondition. This means that there are states in A.pre that may not be mapped into goal states when A is executed. As behaviours can be quite complicated it is not at all clear which states in A.pre the agent must start in to ensure it ends up in $\mathtt{G}_1 \wedge \mathtt{G}_2$ after executing A.

The simplest assumption that can be made in this situation is that executing a behaviour will only make those changes to the agent's state which are needed to get from the preconditions of the behaviour to the postconditions. This assumption is known as the *frame axiom*. Provided this criteria is met we can use it to restrict a behaviour's precondition by conjuncting it with those goal conditions which did not subsume the behaviour's post condition. In Figure 2 A.pre is intersected with $\mathtt{G}_2$. Behaviour D is removed from the TR tree as its postcondition is no longer contained within the states thought to get the agent to the goal. The frame axiom is used again to propagate the $\mathtt{G}_2$ condition through behaviour C but is not needed for B since B.post is contained within $\mathtt{G}_2$.

2.3 Side Effects

The frame axiom does not always hold, especially in complex domains. Primarily, this is because the domain expert who provides the behaviour definitions to the agent cannot always foresee how they will affect the agent's state. If the successful execution of behaviour A from a state $s_0 \in \mathtt{A.pre} \cap \mathtt{G}_i$ terminates in

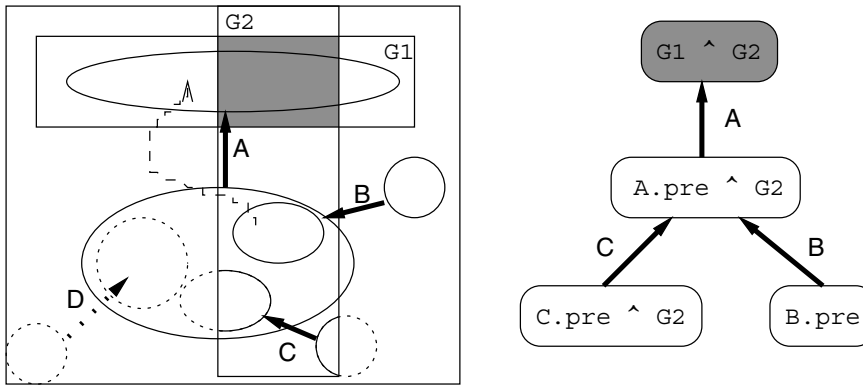


Fig. 2. An example of the frame axiom. Only $\mathtt{G}_1$ subsumes A.post so the second goal condition $\mathtt{G}_2$ must be propagated down the TR tree. The dashed line represents a sequence of steps which violate the frame axiom

a state $s_t \notin \mathtt{G}_i$ we say that executing A when in state s_0 *causes a side effect on* $\mathtt{G}_i$. This is denoted $\mathtt{cause}(s_0, \mathsf{A}, \mathtt{G}_i)$. An example of a side effect is shown as the dashed line in Figure 2.

In order to make effective plans the agent needs to know which states in a behaviour's precondition will cause a certain side effect. Precisely, the agent would need some description of the sets

$$\mathtt{cause}(\mathsf{A}, \mathtt{G}_i) = \{s \in \mathcal{S} : \mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)\}$$

for every behaviour B and every goal condition $\mathtt{G}_i$. In Figure 3 the shaded set labelled cause is the side effect $\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. Since the postcondition of behaviour B falls entirely within this set there is no point considering plans where the agent begins in B.pre. The agent's only option from these states is to execute behaviour B which will terminate at one of the side effect states. By definition, behaviour A will move the agent outside of the set $\mathtt{G}_2$ and hence outside the goal. The only states which will get the agent to the goal using A are those in $\mathsf{A} \wedge \mathtt{G}_2 \wedge \neg\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. This situation is reflected in the new TR tree in the right of the figure.

The simplest approach to constructing the side effect sets is by adding states as they are recognized as causing a side effect. This approach has two problems. Firstly, since the behaviour's policies are being reinforcement learnt as they are used, side effects may be generated due to a bad policy rather than a true side effect of the ideal behaviour. This noise can mean the side effects are over-general which in turn may prevent good planning. The second and more serious difficulty with this method is that only those states already seen to have caused a side effect will be avoided by the agent in the future. If there are regularities across the states causing a side effect we would like the agent to be able exploit them

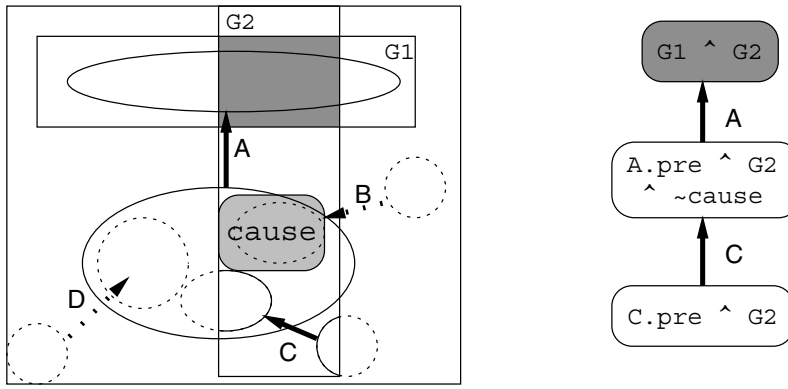


Fig. 3. The side effect set $\texttt{cause}(\mathsf{A}, \mathtt{G}_2)$ is shown as the shaded area labelled `cause` on the left. The revised TR tree is shown on the right

and avoid those states before having to test them explicitly. This is the focus of the next section.

3 Learning Side Effects Using ILP

In his system TRAIL, Benson describes the first application of Inductive Logic Programming to the problem of action model learning [2]. In his model an agent is endowed with actions in the form of TOPs. When these are given to an agent the policies and postconditions of the TOPs are fixed and it is up to the agent to determine satisfactory preconditions. The TRAIL system employs a DINUS-like[4] algorithm to induce the preconditions from examples generated from plan failures, successes and a human teacher. Side effects are recognized as a problem in TRAIL but are only treated lightly through simple statistical methods.

In the RL-TOP system the focus is on learning the policy for each behaviour while the pre- and postconditions are assumed to be correct and fixed. Once the behaviours' policies are learnt sufficiently well side effects become the primary source of an agent's poor performance. Our hypothesis for this paper is that ILP can be used to improve an agent's performance by inducing descriptions for side effects in a manner similar to the way TRAIL uses ILP to learn TOP preconditions.

The ILP learning task will be to construct definitions for the predicates $\texttt{cause}(s, \mathsf{A}, \mathtt{G}_i)$ for various values of A and $\mathtt{G}_i$ as they are required. Once these predicates are learnt the sets $\texttt{cause}(\mathsf{A}, \mathtt{G}_i)$ can be incorporated into an agent's TR tree using the `cause`/3 predicate as a test for a state's membership in a side effect set.

3.1 Examples and Background Knowledge

Recall that any state an agent is in can be uniquely described in terms of the primitive predicates $\mathcal{P} = \{\mathtt{P}_1, \ldots, \mathtt{P}_m\}$. In order to record the history of an agent, the first argument of each $\mathtt{P}_i$ will hold a unique state identifier such as the number of steps the agent has taken since the start of some learning task. High-level predicates will also be modified in this way.

Given a sequence of agent steps $s_0, \ldots, s_n$, examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$ can be found by locating a subsequence $s_a, \ldots, s_b$ such that for all $k = a, \ldots, b-1$ each state $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ and $s_b \notin \mathtt{G}_i$. Since each of the states in the subsequence is one from which A was executed and resulted in $\mathtt{G}_i$ no longer holding, each $\mathtt{cause}(s_k, \mathsf{A}, \mathtt{G}_i)$ is a positive example of the side effect. To get negative examples we need to find states for which the execution of A resulted in $\mathtt{G}_i$ holding true in $\mathsf{A}.\mathtt{post}$. All the states in a subsequence $s_a, \ldots, s_b$ such that $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ for $k = a, \ldots, b-1$ and $s_b \in \mathsf{A}.\mathtt{post} \wedge \mathtt{G}_i$ are negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$.

As an illustration a hypothetical agent history is shown in Table 1. States 2,3 and 4 are positive examples of the side effect $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$ while state 6 is a negative example.

Table 1. An example history trace using the TR tree from Figure 2. Steps marked with '+' or '-' are positive and negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$, respectively

Step	Primitive Description	High-level Predicates	Behaviour	Action
0	$\mathtt{P}_1(0,\ldots) \wedge \ldots \wedge \mathtt{P}_m(0,\ldots)$	$\mathsf{B}.\mathtt{pre}(0)$	B	a_3
1	$\mathtt{P}_1(1,\ldots) \wedge \ldots \wedge \mathtt{P}_m(1,\ldots)$	$\mathsf{B}.\mathtt{pre}(1)$	B	a_7
+2	$\mathtt{P}_1(2,\ldots) \wedge \ldots \wedge \mathtt{P}_m(2,\ldots)$	$\mathsf{B}.\mathtt{post}(2) \wedge \mathsf{A}.\mathtt{pre}(2) \wedge \mathtt{G}_2(2)$	A	a_3
+3	$\mathtt{P}_1(3,\ldots) \wedge \ldots \wedge \mathtt{P}_m(3,\ldots)$	$\mathsf{A}.\mathtt{pre}(3) \wedge \mathtt{G}_2(3)$	A	a_2
+4	$\mathtt{P}_1(4,\ldots) \wedge \ldots \wedge \mathtt{P}_m(4,\ldots)$	$\mathsf{A}.\mathtt{pre}(4) \wedge \mathtt{G}_2(4)$	A	a_5
5	$\mathtt{P}_1(5,\ldots) \wedge \ldots \wedge \mathtt{P}_m(5,\ldots)$	$\mathsf{A}.\mathtt{pre}(5)$	A	a_3
−6	$\mathtt{P}_1(6,\ldots) \wedge \ldots \wedge \mathtt{P}_m(6,\ldots)$	$\mathsf{A}.\mathtt{pre}(6) \wedge \mathtt{G}_2(6)$	A	a_6
7	$\mathtt{P}_1(7,\ldots) \wedge \ldots \wedge \mathtt{P}_m(7,\ldots)$	$\mathsf{A}.\mathtt{post}(7) \wedge \mathtt{G}_2(7) \wedge \mathtt{G}_1(7)$	A	—

The background knowledge for this learning task should, at the very least, consist of all those primitive predicates used to describe the agent's state in each of the examples of the side effect to be learnt. The high-level predicates true in the example states may also be useful when learning a side effect. Whether or not any additional relations are required will depend on the agent's domain. In the above example $\mathtt{cause}(3, \mathsf{A}, \mathtt{G}_2)$ is a positive instance so at least each $\mathtt{P}_i(3,\ldots)$ should appear as background knowledge whereas the high-level predicates $\mathsf{A}.\mathtt{pre}(3)$ and $\mathtt{G}_2(3)$ may or may not be useful.

3.2 Implementation

There are three parts to the RL-TOP hierarchical reinforcement learning system: an *actor*, a *planner* and a *reflector*.[1] Using the high-level view afforded by the RL-TOPs the planner builds a TR plan tree which is passed on to the actor. The plan tree is essentially a description of sequences of behaviours the actor should perform to reach the goal from various states in the state space.

The actor is the agent's interface with the primitive level of the domain. After using the TR tree to determine which behaviour is most appropriate, the agent executes and reinforces its policy. Side effects the agent encounters while executing a behaviour are indicative of a problem with the actor's TR tree. When they occur the actor keeps a record of positive and negative instances of them and hands these examples to the reflector.

The reflector's role is to induce descriptions of which states cause side effects from the examples handed to it from the actor along with any background knowledge it may have been given. Once a side effect has been learnt its description is passed to the planner. The planner then builds a new TR tree and passes it to the actor closing the learning cycle.

The induction of side effect descriptions in the reflector is performed by the ILP system LIME [9]. The deciding factors used to make this choice were speed, noise resistance and a familiarity with the system.[2] It is important the reflector be reasonably quick since each of the three components of the system run asynchronously. If the reflector runs too slowly the actor and planner will continue controlling the agent poorly defeating the purpose of learning side effects in the first place. Also, the examples given to the reflector suffer from a unusual form of noise. When the agent begins its learning task its policies are random. This means initially that examples of a behaviours' side effects are most likely due to poor execution. As the agent steadily improves its policies the proportion of noisy examples decreases. LIME proved to be quite robust under these conditions in addition to being fast enough for our requirements.

3.3 Batch Learning from Incrementally Generated Examples

Reinforcement learning relies heavily on repetition. In order to learn a control model for a problem an agent must repeatedly attempt to reach a set goal. Side effect examples are therefore generated by the actor incrementally. Since LIME is a batch learner we will briefly discuss a method of converting an incremental learning problem into a batch learning problem.

Each time a new side effect is encountered by the actor a *example pool* for that side effect is created. As examples of the side effect are generated they are added to the pool. Before any induction is performed the pool must contain at least E^{+}_{min} positive and E^{-}_{min} negative examples. This is to stop the reflector

[1] This is the part of the system which looks back, or *reflects*, on the actions and plans of the other two parts

[2] FOIL and PROGOL were also briefly tried as induction engines for the reflector however LIME showed more promise during our early tests

attempting to induce side effects from insufficient data. Once these lower limits are both met E^+_{min} positive and E^-_{min} negative examples are randomly sampled from those in the pool and passed to LIME.

As the reflector can only learn one side effect at a time those side effects with a sufficient number of examples are queued. Each time a side effect is learnt it is placed directly on the back of the queue to be re-examined at a later date. This process is an attempt to spread the reflector's attention evenly over its various tasks. When a side effect is learnt for the second time a choice must be made to keep the new definition or discard it in favour of the old. In this situation both versions are tested for accuracy on all the examples currently in the side effect's pool and the better one is kept.

The pool also has upper limits to the number of examples it can hold: E^+_{max} and E^-_{max} specify the maximum number of positive and negative examples that can be stored. When one of these maxima is reached any further example of the same sign randomly replaces a like example already in the pool.

When a side effect gets used by the planner the agent will avoid the area it defines. This means there will be a sudden change in the distribution of examples for that side effect. In particular, the number of positive examples generated for a side effect tends to drop drastically after it is learnt for the first time. The random replacement of old examples with new ones is an attempt to smooth out any drastic changes in the example distributions.

The implementation of the reflector used in the experiment described in Section 4 had example pools defined by $E^+_{min} = 100$, $E^-_{min} = 1000$, $E^+_{max} = 1000$, and $E^-_{max} = 10000$. Having ten times as many negative examples as positive was made to help prevent overgeneralisation as well as reflect the actual ratio of examples being generated.

4 Experimental Results

Standard, non-hierarchical reinforcement learning techniques, such as Q-learning, can be shown to converge to optimal policies, given some fairly natural assumptions [15]. This convergence, especially for large state spaces, can be very slow. Hierarchical reinforcement learning trades optimality for speed hoping to find good solutions quickly by breaking a monolithic reinforcement learning task into smaller ones and combining their solutions. Early versions of the RL-TOP hierarchical reinforcement learning system did not have a reflector and so could not identify and avoid side effects. It was believed that this was the cause of some of the early system's sub-optimality. The experiment reported in this section was designed to test if adding the ability to recognize side effects meant the RL-TOP system could match the performance of monolithic reinforcement learning in the long term while retaining the speed gained through the hierarchical approach.

4.1 The Mudworld

The test domain for our experiment, dubbed "The Mudworld", was chosen to best illustrate the impact of side effects on the agent's performance. It is small

enough to be monolithically reinforcement learnt in a reasonable amount of time but large enough for hierarchical techniques to show a speed improvement. The other advantage to using this artificial domain is that the side effects the agent will encounter are quite obvious. The side effect descriptions generated by the reflector can therefore be checked for their pertinence.

As seen in Figure 4 the Mudworld lies on a 60-by-60 grid which is partitioned into seven rooms. The agent's task in this domain is to build a control model which will move it from any starting position in Room 2 to any square Room 6 without being muddy. The agent becomes muddy by stepping into the rectangular patch of mud in Room 4. Once this happens the agent can only become clean again by stepping into the square of water in Room 6. If the agent avoids the mud entirely as it moves from Room 2 to Room 6 (via Rooms 1, 4 and 5, for example) the goal condition of being clean in Room 6 is met as soon as the agent is inside the door to Room 6. If the agent moves through the mud (by choosing to go through Rooms 1, 4 and 7) it does not reach a goal state until it steps in the water near the centre of Room 6. This ensures there is a performance penalty of around 20 steps for a bad choice of route from Room 2 to Room 6.

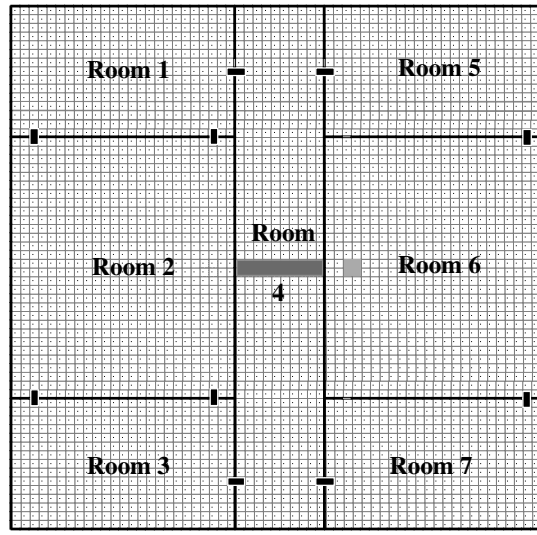


Fig. 4. The Mudworld consists of 7 rooms, connected by doors, on a 60 by 60 grid. The top left of the grid is position (0,0). There is mud in Room 4 (dark grey strip) which can only be cleaned off by the water in Room 6 (light grey square). The agent is started in a random position in Room 2 and its goal is to be clean in Room 6

The agent's primitive state in the Mudworld is described by two primitive predicates: $\texttt{position}(S, X, Y)$ specifying the agent's X and Y coordinate on the grid in state S, and $\texttt{muddy}(S)$ which is true only when the agent is muddy in state S. The agent's primitive actions enable it to move from a grid position to any of its eight adjacent grid positions, walls permitting.

A high-level predicate $\texttt{in\_room}(S, R)$ is defined using the primitive predicate $\texttt{position}/3$ and the "less than" relation $<$. The clause defining all the states in Room 1 is:

$$\texttt{in\_room}(S, 1) \quad \texttt{:-} \quad \texttt{position}(S, X, Y), \quad X < 25, \quad Y < 15.$$

The other rooms are defined similarly. Sixteen RL-TOPs are given to the agent specifying the pre- and postconditions for behaviours which move the agent from one room into an adjacent room. All of these behaviours can be summed up with the following template:

$$\texttt{go}(R_1, R_2).\texttt{pre} = \texttt{in\_room}(R_1)$$
$$\texttt{go}(R_1, R_2).\texttt{post} = \texttt{in\_room}(R_2)$$

The mud in the middle of Room 4 makes planning difficult with the above RL-TOPs. Every plan capable of getting the agent from the left-hand side of the map to the right-hand side must, at some point, pass through Room 4 into either Room 5 or Room 7. Since the only high-level predicate the planner has to describe the agent's state in Room 4 is $\texttt{in\_room}(4)$ it has to make a choice between telling the agent to use $\texttt{go}(4, 5)$ or $\texttt{go}(4, 7)$ at this point in the plan. Neither of these behaviours are ideal. If the agent is in the upper half of Room 4 executing the $\texttt{go}(4, 7)$ behaviour will move the agent through the mud causing a side effect. Similarly, if the agent is the the lower half of Room 4 the $\texttt{go}(4, 5)$ behaviour will cause a side effect.

In a sense, the planner's high-level representation of the agent's state is not rich enough to come up with a plan that will always avoid getting muddy. The job of the reflector then is to define new high-level terms which give the planner a better description of the agent's world. The definitions that will do this job in the above situation are:

$$\texttt{cause}(S, \texttt{go}(4, 5), \texttt{muddy}) \quad :- \quad \texttt{position}(S, \_, Y), \quad 30 < Y.$$
$$\texttt{cause}(S, \texttt{go}(4, 7), \texttt{muddy}) \quad :- \quad \texttt{position}(S, \_, Y), \quad Y < 29.$$

For comparison, Figure 5 shows a set of clauses the reflector had generated for the planner at the end of one of experiments described in the next section. As can be seen the last two clauses in the figure are quite similar to the ideal definitions given above although the last clause is a little over-general. The first two clauses in the figure describe "causes of causes". For example, the first clause states that if an agent is going out of Room 1 it will be in the side effect area for getting `muddy` when executing $\texttt{go}(\texttt{room(4)}, \texttt{room(7)})$.

```
cause(A, go(room(1),_), cause(go(room(4),room(7)),muddy)) :-
        in_room(A, room(1)).
cause(A, go(room(3),_), cause(go(room(4),room(5)),muddy)) :-
        in_room(A, room(3)).
cause(A, go(B,room(5)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        greater_than(A, C, 30).
cause(A, go(B,room(7)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        less_than(A, C, 38).
```

Fig. 5. Output from the reflector after one run of the experiment. The planner uses these definitions to rebuild the TR tree

4.2 Experimental Setup

A standard measure of agent performance in reinforcement learning is *trial length*. In this experiment a trial begins with the agent starting clean and at some random position in Room 2. The trial comes to an end when the agent is clean and in Room 6. The number of primitive actions, or *steps*, the agent takes to reach the goal from the starting state is the trial's length. After each trial the agent updates its policies, behaviours and plans if necessary and is restarted. In order to assess an agent's long term performance it needs to be allowed to improve over many trials. We therefore define a *run* to be a sequence of 10000 consecutive trials.

Our Mudworld experiment compares three different *approaches*: a monolithic reinforcement learning task in which a single policy is learnt for the entire Mudworld (the MRL approach), an RL-TOPs-based hierarchical reinforcement learning approach that does not use a reflector (the RHRL approach), and the same behaviour-based approach incorporating a reflector to learn side effects (the RHRL+R approach). The testing of each approach consists of twenty independent runs. When we speak of *average trial length* the averaging is done across these twenty runs.

The results of this experiment are summarized in Figure 6. The agent's primitive actions are considered to be atomic and so are used as a unit of time along the horizontal axis. Each curve shows how the average trial length changes over one run of an approach. Each point's horizontal coordinate represents when one (average) trial ends and the next begins. Thus, the right-hand end of each curve gives an indication of how long it takes each approach to perform 10000 trials. The vertical value of each point would ideally show the average trial length but this value is far too erratic, especially in the early stages of a run. To get a clearer, qualitative picture the curve is smoothed by averaging the average trial

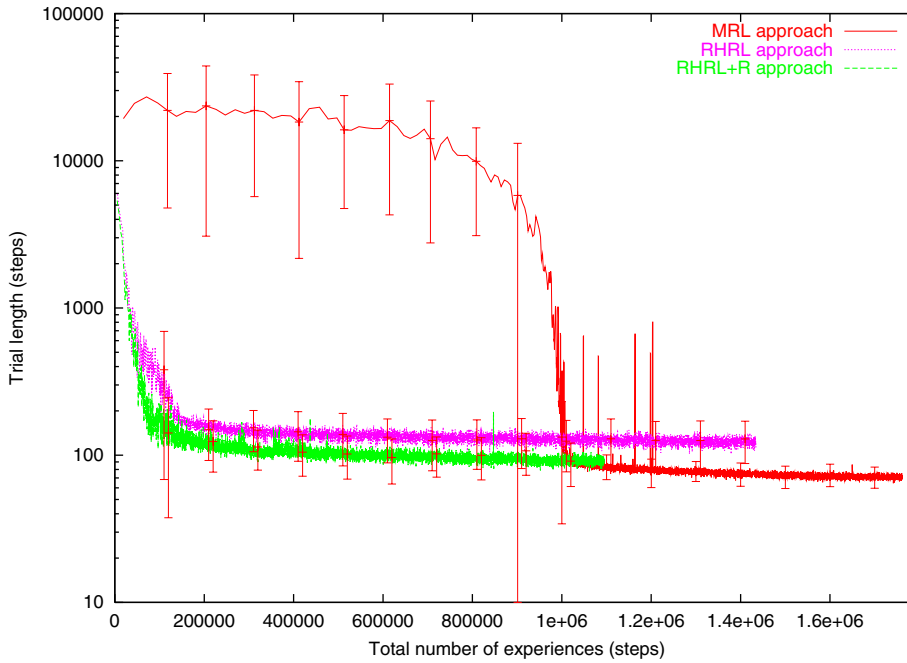


Fig. 6. A comparison of three approaches to learning a control model for a Mudworld agent. Each curve shows how average trial length decreases as the agent becomes more experienced over time

length over a 100-point window about each point. For further clarity the error bars, representing one standard deviation, are only shown every 100000 points.

4.3 Analysis and Conclusions

The most interesting features in Figure 6, in light of our hypothesis, are the relative performances of the three approaches at the end of a run and their convergence rates.

The two hierarchical learning approaches, RHRL and RHRL+R, both show a large improvement in convergence rate compared to the MRL approach which took almost five times as many steps to reach the same level of performance. While the addition of a reflector to the RL-TOPs system was not detrimental to its convergence rate it is unclear from the current data whether or not it offers any improvement.

Figure 7 compares the performance of each of the three approaches after they have converged. Each point on the graph shows the *mean final trial length* for each run in the experiment (20 runs for each of the MRL, RHRL and RHRL+R approaches). It is calculated by averaging the number of steps per trial over the

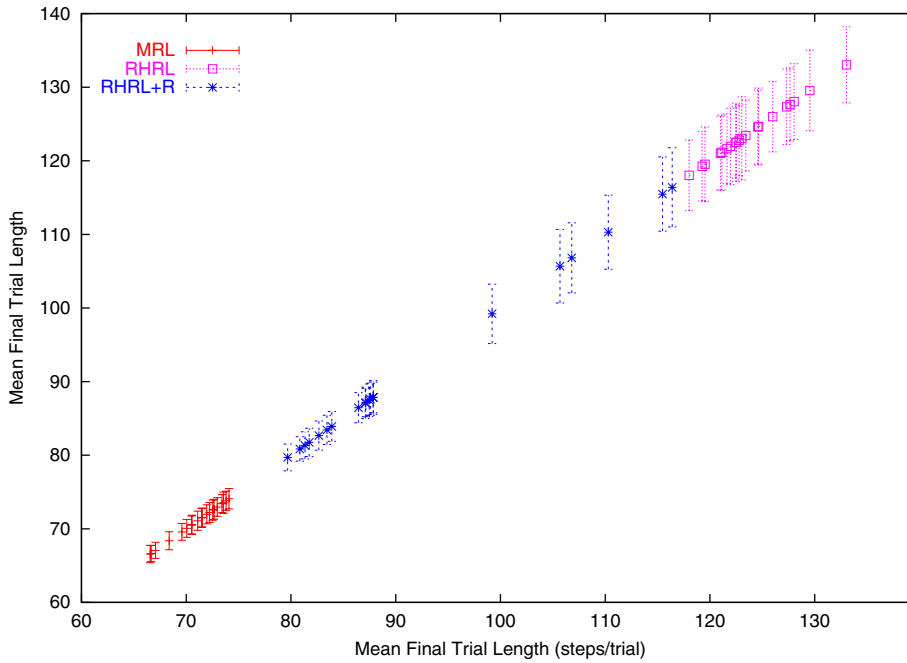


Fig. 7. The mean final trial length (averaged over the final 500 trials) shown for each approach's 20 runs. All of the MRL runs have a final average between 65 and 75 steps per trial. RHRL and RHRL+R runs lie in the ranges 80–117 and 118–135 respectively. The error bars show 99% confidence intervals

last 500 trials in a run. We are assuming that there is no significant improvement taking place between these final trials and therefore the mean is essentially over 500 independent trials of a well-trained agent.

As expected, the performance of monolithic reinforcement learning (MRL) was significantly better than either of the other two approaches. All of its twenty runs had final trial lengths between 65 and 75 steps per trial while the best RHRL+R and RHRL runs were at 80 and 118 steps per trial respectively. A striking feature of the graph is the spread of final trial lengths for the RHRL+R approach. Fourteen of the RHRL+R runs had final trial lengths between 80 and 90 steps per trial while three of its runs were statistically indistinguishable from the best RHRL run at 99% significance. The spread of results for the RHRL+R approach is due to the reflector not always inducing good definitions for side effects. We will be looking into ways of addressing this problem in the future.

As can be seen, there is some sacrifice of optimality for better convergence speed. Even the best RHRL+R run had a higher mean final trial length than the MRL approach. This is because in the reinforcement learning in the RHRL+R approach improves agent policy locally whereas the MRL can optimize over the

entire problem. For example, if the agent starts in the middle left of Room 2 the best policy for the long-term goal (being in Room 6 without being muddy) is to head diagonally towards the door closest to Room 4. However, there is a different best policy for the short-term goal (moving into Room 1). Namely, going through the leftmost door. These inefficiencies are slight compared to the large gain in convergence speed afforded by RHRL+R.

We consider these results to be evidence supporting our hypothesis: that using ILP to predict side effects can improve the long-term performance of the RL-TOP hierarchical reinforcement learning system. This performance is, more often than not, much better than the long-term performance of hierarchical reinforcement learning without a reflector and is comparable to the performance of monolithic reinforcement learning. Furthermore, both heirarchical reinforcement learning methods show a drastic improvement over standard reinforcement in the time it takes to converge to a good policy.

5 Conclusions, Discussion, and Related Work

The work presented in this paper is synthesis of ideas from several areas in artificial intelligence. By representing an agent's state symbolically RL-TOPs are able to forge a link between reinforcement learning and planning. This link is not perfect, however, since the acting and planning sides of an RL-TOP agent represent its world at different levels of abstraction. Watching what happens at these different levels simultaneously is essential to an agent's overall improvement. Inductive logic programming realizes this improvement by allowing the agent to construct new state abstractions which can be used to avoid side effects during the execution of its plans.

This is not the first time ILP has been used in conjunction with reinforcement learning or planning. In [5], Džeroski et al introduce relational reinforcement learning and show that a simple planning task – block's world – can be solved in their framework. The use of ILP in their work differs from that presented here. Rather than inducing descriptions for subsets of the state space as we do here, relational reinforcement learning uses ILP techniques to help learn a Q-tree, a variation of classic reinforcement learning's Q-function.

Inductive and analytic learning algorithms have been used in the past to improve planning and problem-solving. A good overview of work in this area can be found in [8]. However, the focus of much of this is on learning how to improve search heuristics used to generate plans which will take an agent from its current state to a goal state. When generating TR trees in the RL-TOPs system, the planner does not assume the agent is in any particular starting state. Instead, it generates many plans which lead to the goal from various parts of the state space. The aim of this search tree is to be as exhaustive as possible so learning heuristics to prune the search would not be useful in the present system.

Early work in action-model learning in planning, such as Shen's LIVE system [14] and Gil's EXPO system [6], used inductive learning algorithms to refine models of action pre- and postconditions. Their work is restricted to domains

in which the agent's actions must be atomic and deterministic. These assumptions cannot hold in our RL-TOP framework. Firstly, our agent's behaviours are durative since they are executed by following policies involving many atomic actions. Also, these behaviours are inherently non-deterministic since they are being continually improved by reinforcement learning techniques.

As mentioned in Section 3 our system draws inspiration from Benson's TRAIL system for learning action models for autonomous agents in which ILP is used to find preconditions for TOPs. Benson's work addresses some of the short-comings in Shen and Gil's systems as actions which are non-deterministic and durative (instead of atomic) can be modeled in TRAIL.

Unlike all of these systems that combine learning and symbolic planning, the basic planning unit in our system, the RL-TOP, becomes more effective by improving the execution of its behaviour. Since the pre- and postconditions of each behaviour are given in advance and fixed we have adapted some of the ILP ideas found in Benson's work to the problem of inducing side effect definitions. The result is an agent with finely-tuned, reusable, high level actions that enable it to move between fixed, user-defined portions of its state space. Each action's collection of side effects describes how subsets of its pre- and postconditions are related by its policy. With this more detailed view of its actions the agent is able to plan their use more effectively.

The ILP system LIME [9] was chosen to implement the side effect learning in our system due to its noise handling ability and speed. One difficultly with this choice that had to be overcome was that of using a batch learner in an incremental setting. Our fairly straightforward solution proposed in this paper was to pool examples as they were presented incrementally. Once enough examples were collected they could be passed onto LIME as a batch.

The noise resistant, incremental ILP system HILLARY [7] was brought to our attention some time after we were successfully using our example pool approach with LIME. Once we have a running version of HILLARY we hope to compare its performance as a reflector against our present setup.

Other future work will include applying side effect learning to domains more complicated than the Mudworld task presented here. Our long term goal and original motivation for side effect learning is the "Learning to Fly" behavioral cloning task [13]. Although the RL-TOP system has had some success with learning to fly, finding an ILP system capable of handling the complexity of the side effects involved will not be easy. We will also be investigating whether side effect learning is relevant to other hierarchical reinforcement learning systems (eg, [16], [11], [3]).

6 Acknowledgements

The authors would like to thank several people for their help during the writing of this paper. Mike Bain helped clarify some of the ideas during the early stages of this paper and Eric McCreath was always happy to offer assistance with his ILP system LIME. The experiments performed for this paper would have

been much more time-consuming were it not for Waleed Kadous's set up and support of CSE's Beowulf cluster and Virginia Wheway's advice on some of the statistical analysis. The anonymous reviewers also provided us with a number of good references and suggestions.

The first author was supported by an Australian Postgraduate Award while working on this paper.

References

[1] *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998.
[2] Scott Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Department of Computer Science, Stanford University, 1996.
[3] Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].
[4] S. Džeroski, S. Muggleton, and S. Russel. PAC learnability of determinate logic programs. In *Proceeding of the Fifth ACM Workshop on Computational Learning Theory*, pages 128–135, 1992.
[5] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *Proceedings of the 8th International Workshop on Inductive Logic Programming*, pages 11–22, 1998.
[6] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the 11th International Workshop on Machine Learning*, 1994.
[7] Wayne Iba, James Wogulis, and Pat Langley. Trading off simplicity and coverage in incremental concept learning. In *Proceedings of the 5th International Conference on Machine Learning*, pages 73–79, 1988.
[8] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.
[9] E. McCreath and A. Sharma. Lime: A system for learning relations. In *The 9th International Workshop on Algorithmic Learning Theory*. Springer-Verlag, October 1998.
[10] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.
[11] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference*, 1998.
[12] Malcolm R. K. Ryan and Mark D. Pendrith. RL-TOPS: An architecture for modularity and re-use in reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].
[13] Malcolm R. K. Ryan and Mark Reid. Learning to fly: An application of hierarchical reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*. Morgan Kaufmann, (to appear).
[14] Wei-Min Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 12:143–156, 1993.
[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[16] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

Using ILP to Improve Planning in Hierarchical Reinforcement Learning

Mark Reid and Malcolm Ryan

School of Computer Science and Engineering, University of New South Wales
Sydney 2052, Australia
{mreid,malcolmr}@cse.unsw.edu.au

Abstract. Hierarchical reinforcement learning has been proposed as a solution to the problem of scaling up reinforcement learning. The RL-TOPs Hierarchical Reinforcement Learning System is an implementation of this proposal which structures an agent's sensors and actions into various levels of representation and control. Disparity between levels of representation means actions can be misused by the planning algorithm in the system. This paper reports on how ILP was used to bridge these representation gaps and shows empirically how this improved the system's performance. Also discussed are some of the problems encountered when using an ILP system in what is inherently a noisy and incremental domain.

1 Introduction

Reinforcement learning [15] has been studied for many years now as approach to learning control models for robot or software agents. It works superbly on small, low-dimensional domains but has trouble scaling up to larger, more complex, problems. These larger problems have correspondingly larger state spaces which means random exploration and reward back-propagation – the key techniques in reinforcement learning – are much less effective.

Hierarchical reinforcement learning has been proposed as a solution to this lack of scalability and comes in several forms ([12], [16], [11], [3]). The overarching idea is to break up a large reinforcement learning task into smaller subtasks, find policies for the subtasks, then finally recombine them into a solution for the original, larger problem. The approach examined in this paper, the RL-TOPs Hierarchical Reinforcement Learning System[13], represents the agent's state symbolically at various levels of abstraction. This allows for a unique synthesis of reinforcement learning and symbolic planning.

Low-level state representation is ideal when an agent requires a fine-grained view of its world (eg, motor control in robot walking and balancing). In order to make larger task tractable, however, a higher level of abstraction is sometimes required (eg, moving the robot through several rooms in an office block). In the RL-TOPs system the high-level representation of a problem is provided by a domain expert who defines coarse-grained actions and states for the agent. The

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 174–190, 2000.

low-level implementation of the coarse actions are left to the agent to invent using reinforcement learning.

The high-level states and actions do not always convey every relevant feature of an agent's state space to the planning side of the RL-TOP system. The work presented in this paper shows that by examining the interplay between the agent's low-level and high-level actions, new high-level features can be constructed using a standard ILP algorithm.

The paper is organized as follows. Section 2 gives an overview of the RL-TOP system and the planning problem motivating the use of ILP to learn new state space features. Section 3 details the transformation of the planning problem into an ILP learning task as well as outlining a method for converting a batch learning algorithm into an incremental learner. Finally, Section 4 describes a simple domain used to test the performance of the RL-TOPs system augmented with an ILP system. An experiment comparing the performance of three reinforcement learning approaches on this domain is then analyzed.

2 Reinforcement-Learnt Teleo-Operators

The key concept in the hierarchical reinforcement system presented in this paper is the *Reinforcement-Learnt Teleo-OPerator* or RL-TOP [12]. It is a synthesis of ideas from planning and reinforcement learning. Like Nilsson's teleo-operators (TOPs) [10], RL-TOPs define agent behaviours by symbolically describing their preconditions and effects. The main advantage RL-TOPs have over standard TOPs is that the agent does not need hard-coded instructions on how to carry out each of its behaviours. Instead, the problem of moving from the set of states defined by an RL-TOP's precondition to the states defined by its intended effect is treated as a reinforcement learning task. This allows a large reinforcement learning task to be broken down into several easier ones which are combined together by a symbolic planner.

The use of RL-TOPs in the system presented in this paper is analogous to the use of *options* as described by Sutton et al in [15], the *Q nodes* of Dietterich's MAXQ system [3] as well as *abstract actions* and *macro-actions* described elsewhere in the literature. The RL-TOP approach distinguishes itself from these other systems by emphasizing the symbolic representation of agent behaviours.

2.1 Definitions and Notation

At any point in time an agent is assumed to be in some *state* $s \in \mathcal{S}$. To move from one state to another an agent has a finite number of *actions* $\mathcal{A} = \{a_1, \ldots, a_k\}$ which are functions from $\mathcal{S}$ to $\mathcal{S}$. Actions need not be deterministic.

Given a set of *goal states* $\mathtt{G} \subset \mathcal{S}$ a reinforcement learning task requires an agent to come up with a *policy* $\pi : \mathcal{S} \rightarrow \mathcal{A}$. A *good* policy is one that can take an agent from any initial state, $s_0 \in \mathcal{S}$, through a sequence of steps, $s_0, \ldots, s_n$, such that $s_n \in \mathtt{G}$. A *step* is made from s_i to s_{i+1} by applying the action $\pi(s_i)$

to the state s_i yielding s_{i+1}. An *optimal* policy is one that for any initial state generates the shortest possible sequence of steps to a goal state.

An RL-TOP or *behaviour* B consists of three components, a *precondition*, a *policy*, and a *postcondition* (or effect). The precondition, denoted B.pre, is a set of states in which the behaviour is applicable. The postcondition, B.post, is another set of states which define a behaviour's intended effect. An agent's behaviour is executed by using its policy, $\mathsf{B}.\pi$, to move from state to state until the agent is no longer in that behaviour's precondition. If, when the execution of the behaviour terminates, the agent is in B.post the behaviour was said to be *successful*. Behaviours are given to the agent by defining its pre- and post-conditions. Finding good policies for each behaviour then becomes a standard reinforcement learning task – steps that lead to a successful execution of a behaviour are rewarded while steps that leave B.pre without ending up in B.post are punished. The implementation details of this reinforcement learning is not sufficiently pertinent to warrant discussion in this paper. We point the interested reader to [13] and [12].

A finite set of predicates, $\mathcal{P} = \{\mathtt{P}_1, \ldots, \mathtt{P}_m\}$, is called *primitive* if every state $s \in \mathcal{S}$ can be uniquely identified with a conjunction of ground instances of these predicates. A primitive set of predicates allows subsets of the state space to be described by new predicates defined in terms of disjunctions of conjunctions built from elements of $\mathcal{P}$. We call these non-primitive predicates *high-level* predicates. In the remainder of this discussion it will be assumed that goals, preconditions, postconditions and other subsets of $\mathcal{S}$ are defined by high-level predicates.

2.2 Planning and the Frame Axiom

The symbolic representation of behaviours' preconditions and postconditions provides the link between planning and reinforcement learning. As sets they define small reinforcement learning tasks and symbolically it becomes possible to build *Teleo-Reactive (TR) plan trees*. We will briefly explain their construction and limitations through a few examples. A more detailed exposition can be found in [2].

Suppose an agent has a goal G and four behaviours: A, B, C, and D. The left-hand side of Figure 1 is an abstract representation of the agent's state space showing the set of goal states as a shaded rectangle and behaviours as labelled arrows connecting their pre- and postcondition sets.

Since, as the figure shows, that A.post $\subset$ G, it must be the case if the agent is in A.pre and successfully executes behaviour A it will achieve the goal G. It is important to note that this subset test is actually performed in the planning algorithm by testing if the high-level predicate for G subsumes that for A.pre. The problem of reaching a state in G has now been reduced to getting to a state in G *or* getting to a state in A.pre and executing behaviour A. This process is repeated with A.pre as the new goal and B.pre $\cup$ C.pre $\cup$ D.pre are then found to be states from which it is possible to achieve the goal G. The TR tree on the right of the figure shows which actions the agent needs to successfully execute to reach the goal. For example, if the agent state is in the set defined by D.pre it

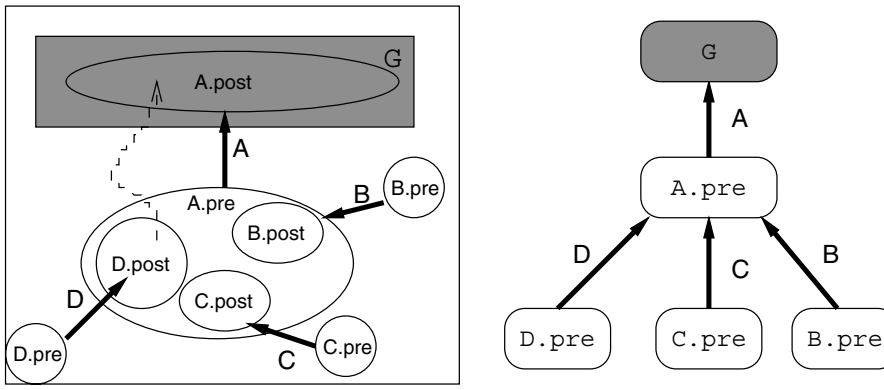


Fig. 1. The diagram on the left shows four behaviours and a goal in a state space. On the right is the corresponding TR plan tree. The dashed line represents a sequence of steps which successfully execute behaviour A

needs to execute behaviour D followed by behaviour A. If an agent moves from a node of a TR tree into a node which is not the one that was expected a *plan failure* is said to have occurred.

The above process for generating TR trees works if the goal state completely subsumes some behaviour's postcondition. The goal in Figure 2, however, is a conjunction of two predicates, $\mathtt{G}_1 \wedge \mathtt{G}_2$, and only one of them, $\mathtt{G}_1$, subsumes A's postcondition. This means that there are states in A.pre that may not be mapped into goal states when A is executed. As behaviours can be quite complicated it is not at all clear which states in A.pre the agent must start in to ensure it ends up in $\mathtt{G}_1 \wedge \mathtt{G}_2$ after executing A.

The simplest assumption that can be made in this situation is that executing a behaviour will only make those changes to the agent's state which are needed to get from the preconditions of the behaviour to the postconditions. This assumption is known as the *frame axiom*. Provided this criteria is met we can use it to restrict a behaviour's precondition by conjuncting it with those goal conditions which did not subsume the behaviour's post condition. In Figure 2 A.pre is intersected with $\mathtt{G}_2$. Behaviour D is removed from the TR tree as its postcondition is no longer contained within the states thought to get the agent to the goal. The frame axiom is used again to propagate the $\mathtt{G}_2$ condition through behaviour C but is not needed for B since B.post is contained within $\mathtt{G}_2$.

2.3 Side Effects

The frame axiom does not always hold, especially in complex domains. Primarily, this is because the domain expert who provides the behaviour definitions to the agent cannot always foresee how they will affect the agent's state. If the successful execution of behaviour A from a state $s_0 \in \mathtt{A.pre} \cap \mathtt{G}_i$ terminates in

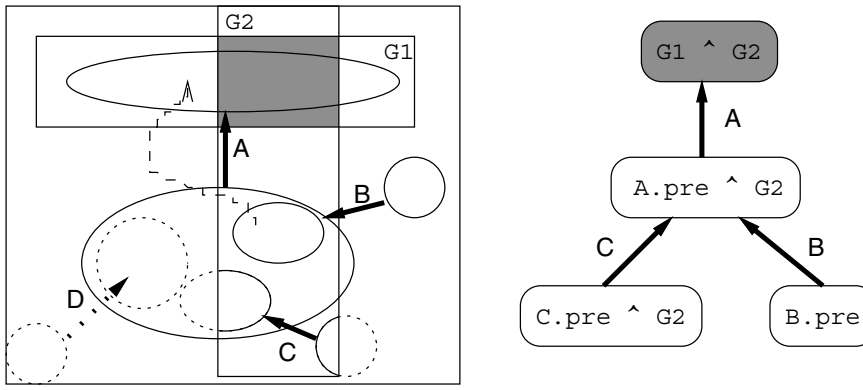


Fig. 2. An example of the frame axiom. Only $\mathtt{G}_1$ subsumes A.post so the second goal condition $\mathtt{G}_2$ must be propagated down the TR tree. The dashed line represents a sequence of steps which violate the frame axiom

a state $s_t \notin \mathtt{G}_i$ we say that executing A when in state s_0 *causes a side effect on* $\mathtt{G}_i$. This is denoted $\mathtt{cause}(s_0, \mathsf{A}, \mathtt{G}_i)$. An example of a side effect is shown as the dashed line in Figure 2.

In order to make effective plans the agent needs to know which states in a behaviour's precondition will cause a certain side effect. Precisely, the agent would need some description of the sets

$$\mathtt{cause}(\mathsf{A}, \mathtt{G}_i) = \{s \in \mathcal{S} : \mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)\}$$

for every behaviour B and every goal condition $\mathtt{G}_i$. In Figure 3 the shaded set labelled cause is the side effect $\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. Since the postcondition of behaviour B falls entirely within this set there is no point considering plans where the agent begins in B.pre. The agent's only option from these states is to execute behaviour B which will terminate at one of the side effect states. By definition, behaviour A will move the agent outside of the set $\mathtt{G}_2$ and hence outside the goal. The only states which will get the agent to the goal using A are those in $\mathsf{A} \wedge \mathtt{G}_2 \wedge \neg\mathtt{cause}(\mathsf{A}, \mathtt{G}_2)$. This situation is reflected in the new TR tree in the right of the figure.

The simplest approach to constructing the side effect sets is by adding states as they are recognized as causing a side effect. This approach has two problems. Firstly, since the behaviour's policies are being reinforcement learnt as they are used, side effects may be generated due to a bad policy rather than a true side effect of the ideal behaviour. This noise can mean the side effects are over-general which in turn may prevent good planning. The second and more serious difficulty with this method is that only those states already seen to have caused a side effect will be avoided by the agent in the future. If there are regularities across the states causing a side effect we would like the agent to be able exploit them

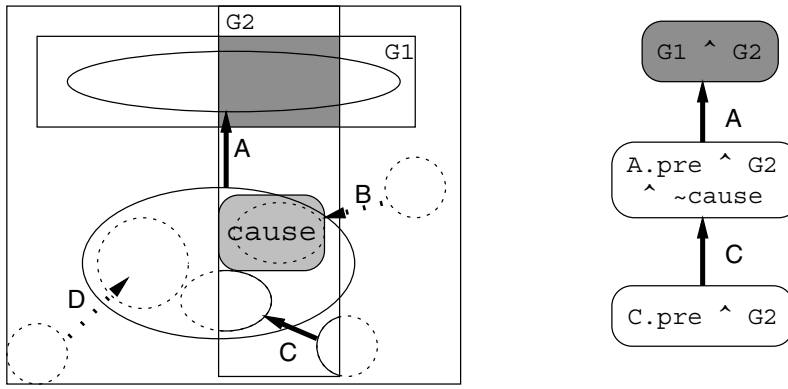


Fig. 3. The side effect set $\texttt{cause}(\mathsf{A}, \mathtt{G}_2)$ is shown as the shaded area labelled `cause` on the left. The revised TR tree is shown on the right

and avoid those states before having to test them explicitly. This is the focus of the next section.

3 Learning Side Effects Using ILP

In his system TRAIL, Benson describes the first application of Inductive Logic Programming to the problem of action model learning [2]. In his model an agent is endowed with actions in the form of TOPs. When these are given to an agent the policies and postconditions of the TOPs are fixed and it is up to the agent to determine satisfactory preconditions. The TRAIL system employs a DINUS-like[4] algorithm to induce the preconditions from examples generated from plan failures, successes and a human teacher. Side effects are recognized as a problem in TRAIL but are only treated lightly through simple statistical methods.

In the RL-TOP system the focus is on learning the policy for each behaviour while the pre- and postconditions are assumed to be correct and fixed. Once the behaviours' policies are learnt sufficiently well side effects become the primary source of an agent's poor performance. Our hypothesis for this paper is that ILP can be used to improve an agent's performance by inducing descriptions for side effects in a manner similar to the way TRAIL uses ILP to learn TOP preconditions.

The ILP learning task will be to construct definitions for the predicates $\texttt{cause}(s, \mathsf{A}, \mathtt{G}_i)$ for various values of A and $\mathtt{G}_i$ as they are required. Once these predicates are learnt the sets $\texttt{cause}(\mathsf{A}, \mathtt{G}_i)$ can be incorporated into an agent's TR tree using the `cause`/3 predicate as a test for a state's membership in a side effect set.

3.1 Examples and Background Knowledge

Recall that any state an agent is in can be uniquely described in terms of the primitive predicates $\mathcal{P} = \{\mathtt{P}_1, \ldots, \mathtt{P}_m\}$. In order to record the history of an agent, the first argument of each $\mathtt{P}_i$ will hold a unique state identifier such as the number of steps the agent has taken since the start of some learning task. High-level predicates will also be modified in this way.

Given a sequence of agent steps $s_0, \ldots, s_n$, examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$ can be found by locating a subsequence $s_a, \ldots, s_b$ such that for all $k = a, \ldots, b-1$ each state $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ and $s_b \notin \mathtt{G}_i$. Since each of the states in the subsequence is one from which A was executed and resulted in $\mathtt{G}_i$ no longer holding, each $\mathtt{cause}(s_k, \mathsf{A}, \mathtt{G}_i)$ is a positive example of the side effect. To get negative examples we need to find states for which the execution of A resulted in $\mathtt{G}_i$ holding true in $\mathsf{A}.\mathtt{post}$. All the states in a subsequence $s_a, \ldots, s_b$ such that $s_k \in \mathsf{A}.\mathtt{pre} \wedge \mathtt{G}_i$ for $k = a, \ldots, b-1$ and $s_b \in \mathsf{A}.\mathtt{post} \wedge \mathtt{G}_i$ are negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_i)$.

As an illustration a hypothetical agent history is shown in Table 1. States 2,3 and 4 are positive examples of the side effect $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$ while state 6 is a negative example.

Table 1. An example history trace using the TR tree from Figure 2. Steps marked with '+' or '-' are positive and negative examples of $\mathtt{cause}(s, \mathsf{A}, \mathtt{G}_2)$, respectively

Step	Primitive Description	High-level Predicates	Behaviour	Action
0	$\mathtt{P}_1(0,\ldots) \wedge \ldots \wedge \mathtt{P}_m(0,\ldots)$	$\mathsf{B}.\mathtt{pre}(0)$	B	a_3
1	$\mathtt{P}_1(1,\ldots) \wedge \ldots \wedge \mathtt{P}_m(1,\ldots)$	$\mathsf{B}.\mathtt{pre}(1)$	B	a_7
+2	$\mathtt{P}_1(2,\ldots) \wedge \ldots \wedge \mathtt{P}_m(2,\ldots)$	$\mathsf{B}.\mathtt{post}(2) \wedge \mathsf{A}.\mathtt{pre}(2) \wedge \mathtt{G}_2(2)$	A	a_3
+3	$\mathtt{P}_1(3,\ldots) \wedge \ldots \wedge \mathtt{P}_m(3,\ldots)$	$\mathsf{A}.\mathtt{pre}(3) \wedge \mathtt{G}_2(3)$	A	a_2
+4	$\mathtt{P}_1(4,\ldots) \wedge \ldots \wedge \mathtt{P}_m(4,\ldots)$	$\mathsf{A}.\mathtt{pre}(4) \wedge \mathtt{G}_2(4)$	A	a_5
5	$\mathtt{P}_1(5,\ldots) \wedge \ldots \wedge \mathtt{P}_m(5,\ldots)$	$\mathsf{A}.\mathtt{pre}(5)$	A	a_3
−6	$\mathtt{P}_1(6,\ldots) \wedge \ldots \wedge \mathtt{P}_m(6,\ldots)$	$\mathsf{A}.\mathtt{pre}(6) \wedge \mathtt{G}_2(6)$	A	a_6
7	$\mathtt{P}_1(7,\ldots) \wedge \ldots \wedge \mathtt{P}_m(7,\ldots)$	$\mathsf{A}.\mathtt{post}(7) \wedge \mathtt{G}_2(7) \wedge \mathtt{G}_1(7)$	A	—

The background knowledge for this learning task should, at the very least, consist of all those primitive predicates used to describe the agent's state in each of the examples of the side effect to be learnt. The high-level predicates true in the example states may also be useful when learning a side effect. Whether or not any additional relations are required will depend on the agent's domain. In the above example $\mathtt{cause}(3, \mathsf{A}, \mathtt{G}_2)$ is a positive instance so at least each $\mathtt{P}_i(3,\ldots)$ should appear as background knowledge whereas the high-level predicates $\mathsf{A}.\mathtt{pre}(3)$ and $\mathtt{G}_2(3)$ may or may not be useful.

3.2 Implementation

There are three parts to the RL-TOP hierarchical reinforcement learning system: an *actor*, a *planner* and a *reflector*.[1] Using the high-level view afforded by the RL-TOPs the planner builds a TR plan tree which is passed on to the actor. The plan tree is essentially a description of sequences of behaviours the actor should perform to reach the goal from various states in the state space.

The actor is the agent's interface with the primitive level of the domain. After using the TR tree to determine which behaviour is most appropriate, the agent executes and reinforces its policy. Side effects the agent encounters while executing a behaviour are indicative of a problem with the actor's TR tree. When they occur the actor keeps a record of positive and negative instances of them and hands these examples to the reflector.

The reflector's role is to induce descriptions of which states cause side effects from the examples handed to it from the actor along with any background knowledge it may have been given. Once a side effect has been learnt its description is passed to the planner. The planner then builds a new TR tree and passes it to the actor closing the learning cycle.

The induction of side effect descriptions in the reflector is performed by the ILP system LIME [9]. The deciding factors used to make this choice were speed, noise resistance and a familiarity with the system.[2] It is important the reflector be reasonably quick since each of the three components of the system run asynchronously. If the reflector runs too slowly the actor and planner will continue controlling the agent poorly defeating the purpose of learning side effects in the first place. Also, the examples given to the reflector suffer from a unusual form of noise. When the agent begins its learning task its policies are random. This means initially that examples of a behaviours' side effects are most likely due to poor execution. As the agent steadily improves its policies the proportion of noisy examples decreases. LIME proved to be quite robust under these conditions in addition to being fast enough for our requirements.

3.3 Batch Learning from Incrementally Generated Examples

Reinforcement learning relies heavily on repetition. In order to learn a control model for a problem an agent must repeatedly attempt to reach a set goal. Side effect examples are therefore generated by the actor incrementally. Since LIME is a batch learner we will briefly discuss a method of converting an incremental learning problem into a batch learning problem.

Each time a new side effect is encountered by the actor a *example pool* for that side effect is created. As examples of the side effect are generated they are added to the pool. Before any induction is performed the pool must contain at least E^{+}_{min} positive and E^{-}_{min} negative examples. This is to stop the reflector

[1] This is the part of the system which looks back, or *reflects*, on the actions and plans of the other two parts

[2] FOIL and PROGOL were also briefly tried as induction engines for the reflector however LIME showed more promise during our early tests

attempting to induce side effects from insufficient data. Once these lower limits are both met E^+_{min} positive and E^-_{min} negative examples are randomly sampled from those in the pool and passed to LIME.

As the reflector can only learn one side effect at a time those side effects with a sufficient number of examples are queued. Each time a side effect is learnt it is placed directly on the back of the queue to be re-examined at a later date. This process is an attempt to spread the reflector's attention evenly over its various tasks. When a side effect is learnt for the second time a choice must be made to keep the new definition or discard it in favour of the old. In this situation both versions are tested for accuracy on all the examples currently in the side effect's pool and the better one is kept.

The pool also has upper limits to the number of examples it can hold: E^+_{max} and E^-_{max} specify the maximum number of positive and negative examples that can be stored. When one of these maxima is reached any further example of the same sign randomly replaces a like example already in the pool.

When a side effect gets used by the planner the agent will avoid the area it defines. This means there will be a sudden change in the distribution of examples for that side effect. In particular, the number of positive examples generated for a side effect tends to drop drastically after it is learnt for the first time. The random replacement of old examples with new ones is an attempt to smooth out any drastic changes in the example distributions.

The implementation of the reflector used in the experiment described in Section 4 had example pools defined by $E^+_{min} = 100$, $E^-_{min} = 1000$, $E^+_{max} = 1000$, and $E^-_{max} = 10000$. Having ten times as many negative examples as positive was made to help prevent overgeneralisation as well as reflect the actual ratio of examples being generated.

4 Experimental Results

Standard, non-hierarchical reinforcement learning techniques, such as Q-learning, can be shown to converge to optimal policies, given some fairly natural assumptions [15]. This convergence, especially for large state spaces, can be very slow. Hierarchical reinforcement learning trades optimality for speed hoping to find good solutions quickly by breaking a monolithic reinforcement learning task into smaller ones and combining their solutions. Early versions of the RL-TOP hierarchical reinforcement learning system did not have a reflector and so could not identify and avoid side effects. It was believed that this was the cause of some of the early system's sub-optimality. The experiment reported in this section was designed to test if adding the ability to recognize side effects meant the RL-TOP system could match the performance of monolithic reinforcement learning in the long term while retaining the speed gained through the hierarchical approach.

4.1 The Mudworld

The test domain for our experiment, dubbed "The Mudworld", was chosen to best illustrate the impact of side effects on the agent's performance. It is small

enough to be monolithically reinforcement learnt in a reasonable amount of time but large enough for hierarchical techniques to show a speed improvement. The other advantage to using this artificial domain is that the side effects the agent will encounter are quite obvious. The side effect descriptions generated by the reflector can therefore be checked for their pertinence.

As seen in Figure 4 the Mudworld lies on a 60-by-60 grid which is partitioned into seven rooms. The agent's task in this domain is to build a control model which will move it from any starting position in Room 2 to any square Room 6 without being muddy. The agent becomes muddy by stepping into the rectangular patch of mud in Room 4. Once this happens the agent can only become clean again by stepping into the square of water in Room 6. If the agent avoids the mud entirely as it moves from Room 2 to Room 6 (via Rooms 1, 4 and 5, for example) the goal condition of being clean in Room 6 is met as soon as the agent is inside the door to Room 6. If the agent moves through the mud (by choosing to go through Rooms 1, 4 and 7) it does not reach a goal state until it steps in the water near the centre of Room 6. This ensures there is a performance penalty of around 20 steps for a bad choice of route from Room 2 to Room 6.

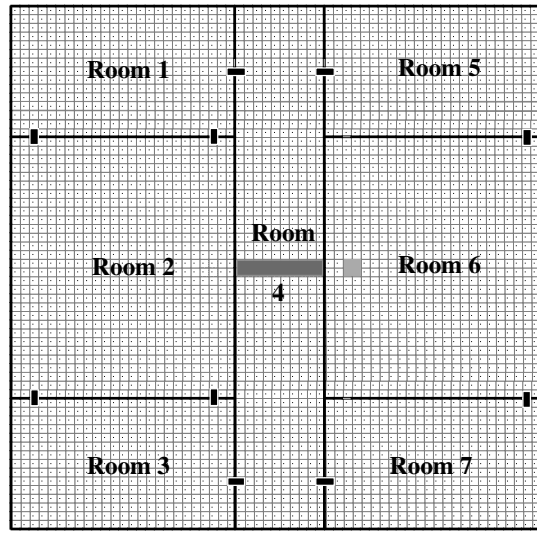


Fig. 4. The Mudworld consists of 7 rooms, connected by doors, on a 60 by 60 grid. The top left of the grid is position (0,0). There is mud in Room 4 (dark grey strip) which can only be cleaned off by the water in Room 6 (light grey square). The agent is started in a random position in Room 2 and its goal is to be clean in Room 6

The agent's primitive state in the Mudworld is described by two primitive predicates: $\texttt{position}(S, X, Y)$ specifying the agent's X and Y coordinate on the grid in state S, and $\texttt{muddy}(S)$ which is true only when the agent is muddy in state S. The agent's primitive actions enable it to move from a grid position to any of its eight adjacent grid positions, walls permitting.

A high-level predicate $\texttt{in\_room}(S, R)$ is defined using the primitive predicate $\texttt{position}/3$ and the "less than" relation $<$. The clause defining all the states in Room 1 is:

$$\texttt{in\_room}(S, 1) \quad \texttt{:-} \quad \texttt{position}(S, X, Y), \quad X < 25, \quad Y < 15.$$

The other rooms are defined similarly. Sixteen RL-TOPs are given to the agent specifying the pre- and postconditions for behaviours which move the agent from one room into an adjacent room. All of these behaviours can be summed up with the following template:

$$\texttt{go}(R_1, R_2).\texttt{pre} = \texttt{in\_room}(R_1)$$
$$\texttt{go}(R_1, R_2).\texttt{post} = \texttt{in\_room}(R_2)$$

The mud in the middle of Room 4 makes planning difficult with the above RL-TOPs. Every plan capable of getting the agent from the left-hand side of the map to the right-hand side must, at some point, pass through Room 4 into either Room 5 or Room 7. Since the only high-level predicate the planner has to describe the agent's state in Room 4 is $\texttt{in\_room}(4)$ it has to make a choice between telling the agent to use $\texttt{go}(4, 5)$ or $\texttt{go}(4, 7)$ at this point in the plan. Neither of these behaviours are ideal. If the agent is in the upper half of Room 4 executing the $\texttt{go}(4, 7)$ behaviour will move the agent through the mud causing a side effect. Similarly, if the agent is the the lower half of Room 4 the $\texttt{go}(4, 5)$ behaviour will cause a side effect.

In a sense, the planner's high-level representation of the agent's state is not rich enough to come up with a plan that will always avoid getting muddy. The job of the reflector then is to define new high-level terms which give the planner a better description of the agent's world. The definitions that will do this job in the above situation are:

$$\texttt{cause}(S, \texttt{go}(4, 5), \texttt{muddy}) \quad :- \quad \texttt{position}(S, \_, Y), \quad 30 < Y.$$
$$\texttt{cause}(S, \texttt{go}(4, 7), \texttt{muddy}) \quad :- \quad \texttt{position}(S, \_, Y), \quad Y < 29.$$

For comparison, Figure 5 shows a set of clauses the reflector had generated for the planner at the end of one of experiments described in the next section. As can be seen the last two clauses in the figure are quite similar to the ideal definitions given above although the last clause is a little over-general. The first two clauses in the figure describe "causes of causes". For example, the first clause states that if an agent is going out of Room 1 it will be in the side effect area for getting `muddy` when executing $\texttt{go}(\texttt{room(4)}, \texttt{room(7)})$.

```
cause(A, go(room(1),_), cause(go(room(4),room(7)),muddy)) :-
        in_room(A, room(1)).
cause(A, go(room(3),_), cause(go(room(4),room(5)),muddy)) :-
        in_room(A, room(3)).
cause(A, go(B,room(5)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        greater_than(A, C, 30).
cause(A, go(B,room(7)), muddy) :-
        in_room(A, B),
        satisfy:not(user:muddy(A)),
        position(A, _, C),
        less_than(A, C, 38).
```

Fig. 5. Output from the reflector after one run of the experiment. The planner uses these definitions to rebuild the TR tree

4.2 Experimental Setup

A standard measure of agent performance in reinforcement learning is *trial length*. In this experiment a trial begins with the agent starting clean and at some random position in Room 2. The trial comes to an end when the agent is clean and in Room 6. The number of primitive actions, or *steps*, the agent takes to reach the goal from the starting state is the trial's length. After each trial the agent updates its policies, behaviours and plans if necessary and is restarted. In order to assess an agent's long term performance it needs to be allowed to improve over many trials. We therefore define a *run* to be a sequence of 10000 consecutive trials.

Our Mudworld experiment compares three different *approaches*: a monolithic reinforcement learning task in which a single policy is learnt for the entire Mudworld (the MRL approach), an RL-TOPs-based hierarchical reinforcement learning approach that does not use a reflector (the RHRL approach), and the same behaviour-based approach incorporating a reflector to learn side effects (the RHRL+R approach). The testing of each approach consists of twenty independent runs. When we speak of *average trial length* the averaging is done across these twenty runs.

The results of this experiment are summarized in Figure 6. The agent's primitive actions are considered to be atomic and so are used as a unit of time along the horizontal axis. Each curve shows how the average trial length changes over one run of an approach. Each point's horizontal coordinate represents when one (average) trial ends and the next begins. Thus, the right-hand end of each curve gives an indication of how long it takes each approach to perform 10000 trials. The vertical value of each point would ideally show the average trial length but this value is far too erratic, especially in the early stages of a run. To get a clearer, qualitative picture the curve is smoothed by averaging the average trial

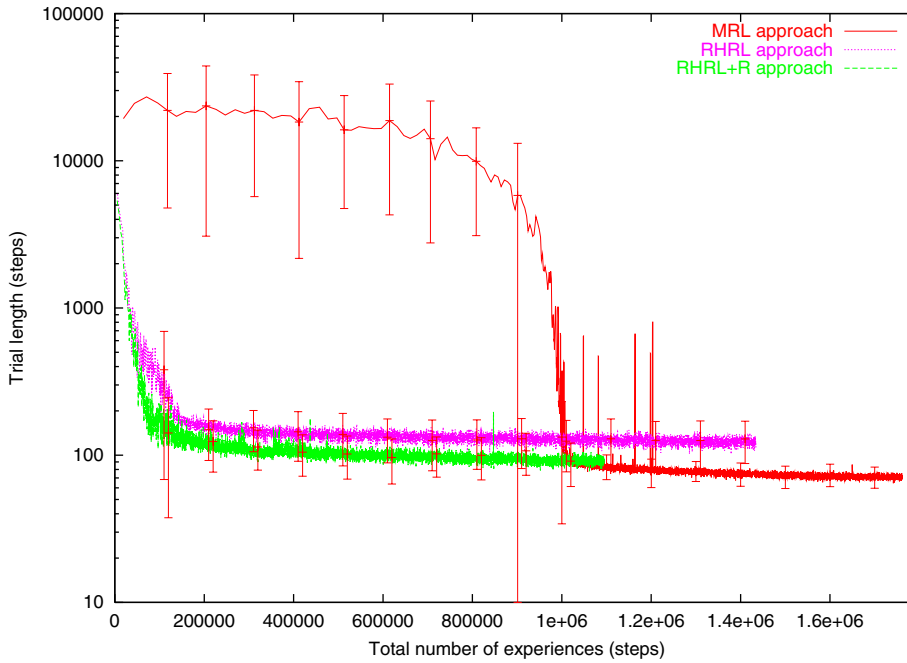


Fig. 6. A comparison of three approaches to learning a control model for a Mudworld agent. Each curve shows how average trial length decreases as the agent becomes more experienced over time

length over a 100-point window about each point. For further clarity the error bars, representing one standard deviation, are only shown every 100000 points.

4.3 Analysis and Conclusions

The most interesting features in Figure 6, in light of our hypothesis, are the relative performances of the three approaches at the end of a run and their convergence rates.

The two hierarchical learning approaches, RHRL and RHRL+R, both show a large improvement in convergence rate compared to the MRL approach which took almost five times as many steps to reach the same level of performance. While the addition of a reflector to the RL-TOPs system was not detrimental to its convergence rate it is unclear from the current data whether or not it offers any improvement.

Figure 7 compares the performance of each of the three approaches after they have converged. Each point on the graph shows the *mean final trial length* for each run in the experiment (20 runs for each of the MRL, RHRL and RHRL+R approaches). It is calculated by averaging the number of steps per trial over the

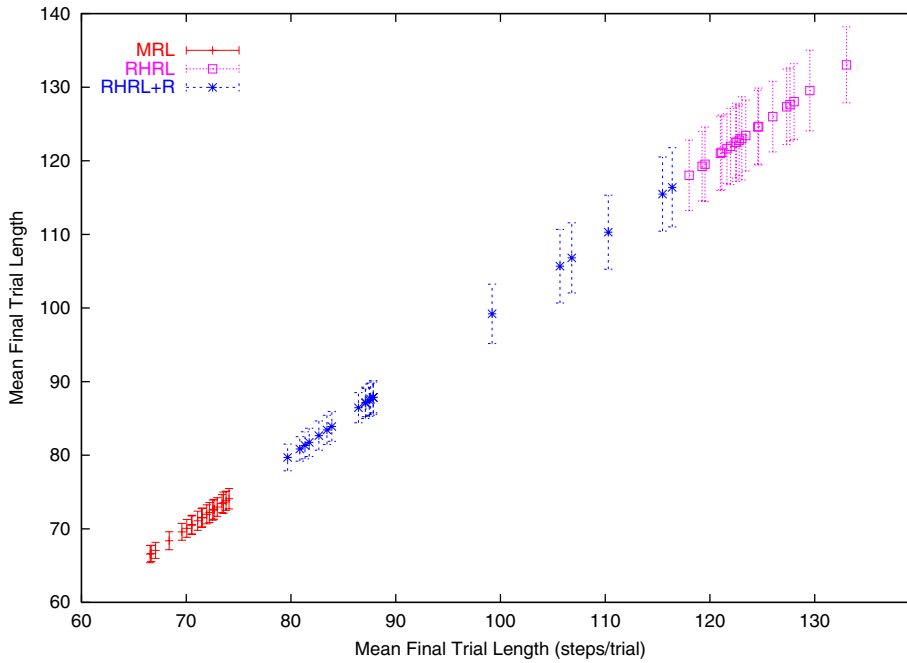


Fig. 7. The mean final trial length (averaged over the final 500 trials) shown for each approach's 20 runs. All of the MRL runs have a final average between 65 and 75 steps per trial. RHRL and RHRL+R runs lie in the ranges 80–117 and 118–135 respectively. The error bars show 99% confidence intervals

last 500 trials in a run. We are assuming that there is no significant improvement taking place between these final trials and therefore the mean is essentially over 500 independent trials of a well-trained agent.

As expected, the performance of monolithic reinforcement learning (MRL) was significantly better than either of the other two approaches. All of its twenty runs had final trial lengths between 65 and 75 steps per trial while the best RHRL+R and RHRL runs were at 80 and 118 steps per trial respectively. A striking feature of the graph is the spread of final trial lengths for the RHRL+R approach. Fourteen of the RHRL+R runs had final trial lengths between 80 and 90 steps per trial while three of its runs were statistically indistinguishable from the best RHRL run at 99% significance. The spread of results for the RHRL+R approach is due to the reflector not always inducing good definitions for side effects. We will be looking into ways of addressing this problem in the future.

As can be seen, there is some sacrifice of optimality for better convergence speed. Even the best RHRL+R run had a higher mean final trial length than the MRL approach. This is because in the reinforcement learning in the RHRL+R approach improves agent policy locally whereas the MRL can optimize over the

entire problem. For example, if the agent starts in the middle left of Room 2 the best policy for the long-term goal (being in Room 6 without being muddy) is to head diagonally towards the door closest to Room 4. However, there is a different best policy for the short-term goal (moving into Room 1). Namely, going through the leftmost door. These inefficiencies are slight compared to the large gain in convergence speed afforded by RHRL+R.

We consider these results to be evidence supporting our hypothesis: that using ILP to predict side effects can improve the long-term performance of the RL-TOP hierarchical reinforcement learning system. This performance is, more often than not, much better than the long-term performance of hierarchical reinforcement learning without a reflector and is comparable to the performance of monolithic reinforcement learning. Furthermore, both heirarchical reinforcement learning methods show a drastic improvement over standard reinforcement in the time it takes to converge to a good policy.

5 Conclusions, Discussion, and Related Work

The work presented in this paper is synthesis of ideas from several areas in artificial intelligence. By representing an agent's state symbolically RL-TOPs are able to forge a link between reinforcement learning and planning. This link is not perfect, however, since the acting and planning sides of an RL-TOP agent represent its world at different levels of abstraction. Watching what happens at these different levels simultaneously is essential to an agent's overall improvement. Inductive logic programming realizes this improvement by allowing the agent to construct new state abstractions which can be used to avoid side effects during the execution of its plans.

This is not the first time ILP has been used in conjunction with reinforcement learning or planning. In [5], Džeroski et al introduce relational reinforcement learning and show that a simple planning task – block's world – can be solved in their framework. The use of ILP in their work differs from that presented here. Rather than inducing descriptions for subsets of the state space as we do here, relational reinforcement learning uses ILP techniques to help learn a Q-tree, a variation of classic reinforcement learning's Q-function.

Inductive and analytic learning algorithms have been used in the past to improve planning and problem-solving. A good overview of work in this area can be found in [8]. However, the focus of much of this is on learning how to improve search heuristics used to generate plans which will take an agent from its current state to a goal state. When generating TR trees in the RL-TOPs system, the planner does not assume the agent is in any particular starting state. Instead, it generates many plans which lead to the goal from various parts of the state space. The aim of this search tree is to be as exhaustive as possible so learning heuristics to prune the search would not be useful in the present system.

Early work in action-model learning in planning, such as Shen's LIVE system [14] and Gil's EXPO system [6], used inductive learning algorithms to refine models of action pre- and postconditions. Their work is restricted to domains

in which the agent's actions must be atomic and deterministic. These assumptions cannot hold in our RL-TOP framework. Firstly, our agent's behaviours are durative since they are executed by following policies involving many atomic actions. Also, these behaviours are inherently non-deterministic since they are being continually improved by reinforcement learning techniques.

As mentioned in Section 3 our system draws inspiration from Benson's TRAIL system for learning action models for autonomous agents in which ILP is used to find preconditions for TOPs. Benson's work addresses some of the short-comings in Shen and Gil's systems as actions which are non-deterministic and durative (instead of atomic) can be modeled in TRAIL.

Unlike all of these systems that combine learning and symbolic planning, the basic planning unit in our system, the RL-TOP, becomes more effective by improving the execution of its behaviour. Since the pre- and postconditions of each behaviour are given in advance and fixed we have adapted some of the ILP ideas found in Benson's work to the problem of inducing side effect definitions. The result is an agent with finely-tuned, reusable, high level actions that enable it to move between fixed, user-defined portions of its state space. Each action's collection of side effects describes how subsets of its pre- and postconditions are related by its policy. With this more detailed view of its actions the agent is able to plan their use more effectively.

The ILP system LIME [9] was chosen to implement the side effect learning in our system due to its noise handling ability and speed. One difficultly with this choice that had to be overcome was that of using a batch learner in an incremental setting. Our fairly straightforward solution proposed in this paper was to pool examples as they were presented incrementally. Once enough examples were collected they could be passed onto LIME as a batch.

The noise resistant, incremental ILP system HILLARY [7] was brought to our attention some time after we were successfully using our example pool approach with LIME. Once we have a running version of HILLARY we hope to compare its performance as a reflector against our present setup.

Other future work will include applying side effect learning to domains more complicated than the Mudworld task presented here. Our long term goal and original motivation for side effect learning is the "Learning to Fly" behavioral cloning task [13]. Although the RL-TOP system has had some success with learning to fly, finding an ILP system capable of handling the complexity of the side effects involved will not be easy. We will also be investigating whether side effect learning is relevant to other hierarchical reinforcement learning systems (eg, [16], [11], [3]).

6 Acknowledgements

The authors would like to thank several people for their help during the writing of this paper. Mike Bain helped clarify some of the ideas during the early stages of this paper and Eric McCreath was always happy to offer assistance with his ILP system LIME. The experiments performed for this paper would have

been much more time-consuming were it not for Waleed Kadous's set up and support of CSE's Beowulf cluster and Virginia Wheway's advice on some of the statistical analysis. The anonymous reviewers also provided us with a number of good references and suggestions.

The first author was supported by an Australian Postgraduate Award while working on this paper.

References

[1] *Proceedings of the 15th International Conference on Machine Learning*. Morgan Kaufmann, 1998.
[2] Scott Benson. *Learning Action Models for Reactive Autonomous Agents*. PhD thesis, Department of Computer Science, Stanford University, 1996.
[3] Thomas G. Dietterich. The maxq method for hierarchical reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].
[4] S. Džeroski, S. Muggleton, and S. Russel. PAC learnability of determinate logic programs. In *Proceeding of the Fifth ACM Workshop on Computational Learning Theory*, pages 128–135, 1992.
[5] Sašo Džeroski, Luc De Raedt, and Hendrik Blockeel. Relational reinforcement learning. In *Proceedings of the 8th International Workshop on Inductive Logic Programming*, pages 11–22, 1998.
[6] Yolanda Gil. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the 11th International Workshop on Machine Learning*, 1994.
[7] Wayne Iba, James Wogulis, and Pat Langley. Trading off simplicity and coverage in incremental concept learning. In *Proceedings of the 5th International Conference on Machine Learning*, pages 73–79, 1988.
[8] Pat Langley. *Elements of Machine Learning*. Morgan Kaufmann, 1996.
[9] E. McCreath and A. Sharma. Lime: A system for learning relations. In *The 9th International Workshop on Algorithmic Learning Theory*. Springer-Verlag, October 1998.
[10] N. J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.
[11] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *Advances in Neural Information Processing Systems 10: Proceedings of the 1997 Conference*, 1998.
[12] Malcolm R. K. Ryan and Mark D. Pendrith. RL-TOPS: An architecture for modularity and re-use in reinforcement learning. In *Proceedings of the 15th International Conference on Machine Learning* [1].
[13] Malcolm R. K. Ryan and Mark Reid. Learning to fly: An application of hierarchical reinforcement learning. In *Proceedings of the 17th International Conference on Machine Learning*. Morgan Kaufmann, (to appear).
[14] Wei-Min Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 12:143–156, 1993.
[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
[16] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

Towards Learning in CARIN-$\mathcal{ALN}$

Céline Rouveirol and Véronique Ventos

Laboratoire de Recherche en Informatique
Bâtiment 490, Université Paris-Sud
91405 - Orsay Cedex (France)
{celine,ventos}@lri.fr

Abstract. In this paper we investigate a new language for learning, which combines two well-known representation formalisms, Description Logics and Horn Clause Logics. Our goal is to study the feasability of learning in such a hybrid description - horn clause language, namely CARIN-$\mathcal{ALN}$ [LR98b], in the presence of hybrid background knowledge, including a Horn clause and a terminological component. After setting our learning framework, we present algorithms for testing example coverage and subsumption between two hypotheses, based on the existential entailment algorithm studied in [LR98b]. While the hybrid language is more expressive than horn clause logics alone, the complexity of these two steps for CARIN-$\mathcal{ALN}$ remains bounded by their respective complexity in horn clause logics.

1 Introduction

Description Logics (DL in the remainder of the paper) have been specifically designed for representing and manipulating hierarchical knowledge. They have been extensively studied wrt two tasks: *least common subsumer* computation and *instance checking*. Although they lack some of the expressive power of Definite Logics (in particular for representing relations between objects), they offer, among other things, interesting connectives (such as cardinality connectives) and a limited form of negation, which can be quite useful in some applications. Moreover, some of them exhibit polynomial complexity in one or both of the tasks quoted above.

Definite Logics offers certainly a greater flexibility for representing complex relations, but the cost for this is high complexity for elementary learning tasks such as subsumption checking. Our goal is to study the feasability of learning in a hybrid description logic - horn clause language in the presence of hybrid background knowledge, including a Horn clause and a terminological component. The added expressive power of such hybrid languages wrt their component languages will enable us to express rich constraints on classes of individuals in the form of a concept hierarchy, while still maintaining the ability to use predicates of any arity and the ability to express arbitrary joins between relations.

We rely on previous theoretical work on a family of such hybrid languages, CARIN [LR98b], which has already been studied in the context of knowledge

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 191–208, 2000.

base verification [LR98a]. The expected pay-off for learning is that we can increase the expressivity of the target concept language without increasing the complexity of the learning process, which remains bounded by the complexity of learning in horn clause logics.

The paper is structured as follows : we first introduce in section 2 the basic notions and notations for DL languages, the CARIN family of languages, and the central inference of CARIN, namely existential entailment. Section 3 introduces our learning framework. Section 4 and 5 respectively detail how we can adapt the existential entailment algorithm in order to test whether a hypothesis covers an example and whether a hypothesis is more general than another. Section 6 studies the complexity of such elementary inferences in our language. Finally, section 7 compares our work to previous approaches and sketches the perspectives of this line of research. Each step will be illustrated by examples inspired from the well-known Michalski's train domain [MMPS94].

2 Knowledge Representation: The CARIN Languages

CARIN is a family of languages, each of them combining a description logic $\mathcal{L}$ with *a set of Horn Clauses*. A CARIN-$\mathcal{L}$ language contains three components. The first is a description logic terminology described using a DL $\mathcal{L}$, the second is a set of Horn clauses and the third is a set of ground facts. The terminology is a set of statements in $\mathcal{L}$ about concepts and roles in the domain. Concepts and roles can also appear as predicates in the antecedents of the Horn clauses and in the ground facts. In contrast, previous hybrid languages (e.g. [DLNS91]) only allow concepts to appear in the Horn clauses. Besides, the variables used in atoms of concepts must appear elsewhere in the rule which is not the case in CARIN.

2.1 Terminological Component: The $\mathcal{ALN}$ Description Logic

Description Logics are a family of knowledge representation formalisms which stem from KL-ONE [Bra78]. Several systems have been built based on DLs (e.g. CLASSIC [BPS94], FLEX [QDBK96]) and they have been used in real-world applications (e.g. CLASSIC in [WWB+93]). In addition, DLs facilitate the use of background knowledge and are more expressive than attribute-value representations. DLs are restrictions of first-order logic different from Horn clauselanguages[1], in which the subsumption computation and its complexity have been deeply studied [DLNN91].

In DLs, there are three kinds of formal objects: *individuals*, sets of individuals called *concepts* which are represented as unary predicates, and relationships between two individuals called *roles*, which are represented as binary predicates. A *concept* is defined as a set of properties satisfied by individuals that are instances of the concept. These properties are expressed by terms that are built

[1] Note that comparing the expressive power of DLs and restrictions of First Order Logic used in ILP is still an open issue [CH94b].

from atomic concepts and roles and from a set of connectives. Concepts are partially ordered by a subsumption relation which expresses the inclusion relation between concepts and is usually based on a standard model-based logical semantics. The basic inference tasks in DLs are determining *subsumption* relations between concepts (this operation enables us to automatically compute the terminology) and checking the membership of an individual in a concept (this operation is called *instance checking*).

In DLs, knowledge is mainly separated into two components: a *terminological component* (refered to as *T-box*), which contains the definition of concepts and an *assertional component* (*A-box*), which contains statements about individuals. The T-Box of a CARIN-$\mathcal{L}$ language will be a classical DL T-box, the ground fact component (section 2.3) will stand for an extension of a classical DL A-box.

The complexity of reasoning in a CARIN-$\mathcal{L}$ knowledge base depends on the expressivity of the terminological components. DLs and their tractability properties vary depending on the set of allowed connectives. For instance, CLASSIC is the most expressive implemented DL for which the subsumption computation is polynomial according to a non-standard semantics. However, this is not enough to guarantee efficient learning in Valiant's sense of PAC-learnability [Val84]. It has been shown in [CH94b, CH94a] that although CLASSIC is not PAC-learnable, some of its subsets are learnable (e.g. C-CLASSIC, k-core-CLASSIC). Because of these results and some complexity results about reasoning in CARIN (see section 6 for a short complexity study), we consider another subset of CLASSIC called $\mathcal{ALN}$.

The terminological language of $\mathcal{ALN}$ is rather simple and is defined using a set $\mathbf{R}$ of atomic roles, a set $\mathbf{P}$ of atomic concepts, the constants $\top$ and $\bot$, and the following syntactic rule (C and D are concepts, P is an atomic concept, R is a atomic role, and n is an integer):

$C, D \rightarrow$	$\top$	the most general concept
	$\mid \bot$	the most specific concept
	$\mid P$	atomic concept
	$\mid \neg P$	negation of atomic concept
	$\mid C \sqcap D$	concept conjunction
	$\mid \forall R.C$	value restriction
	$\mid \geq nR$	cardinality for R (minimum)
	$\mid \leq nR$	cardinality for R (maximum)

This syntactic rule is used to define $\mathcal{ALN}$ *terms*.

Example 1 *Train* $\sqcap \geq 2$ *has_car* $\sqcap \leq 4$ *has_car* $\sqcap \forall$*has_car.Square* $\sqcap \forall$*has_car.*$\leq$ *2 has_load describes all trains that have between two and four cars, and whose cars are square and have two or less loads.*

A terminological component *TC* is composed of T-box statements that have one of the following forms :

1. $A \equiv C$
2. $A \sqsubseteq C$

where A is a concept name and C is a *term* of the $\mathcal{ALN}$ terminological language. A concept name A depends on a concept name A' if A' appears in the definition of A. A set of concept definitions is said acyclic if there is no cycle in the concept name dependency relation. Here, we only consider acyclic concept definitions. We assume that a concept name appears on the left hand side of at most one statement. A concept name A occurring on the left-hand of a statement of type 1 is called a *defined concept* (i.e. A has necessary and sufficient properties). A concept name A occurring on the left-hand of a statement of type 2 is called a *primitive concept* (i.e. A has necessary but not sufficient properties).

Example 2 *The concept definition light_loaded_train* $\equiv$ *train* $\sqcap \geq$ *1 has_car* $\sqcap$ $\forall$*has_car.(*$\geq$ *1 has_load* $\sqcap \leq$ *2 has_load) means that a train is not too loaded if it has more than one car, and that each of its cars contains one or two loads.*

Note that role fillers of individuals can be be precised whitout being enumerated. For instance, we can assert how many objects an individual is related to via some role whitout knowing these objects (e.g. $\geq 2has-car(a)$: the train a has at least two cars), or describe the fillers of a role whitout knowing them (e.g. $\forall has-car : square(a)$: all the cars of the train a are square). The absence of the closed-world assumption enables to support partially known situations. When new knowledge is acquired incomplete information may be gradually refined if the additional knowledge do not contradict past data. The semantics of the terminological component is given via interpretations. Let Δ be the domain and an interpretation function I. The extensions of $\mathcal{ALN}$ terms are computed in the following manner:

- The extension of an atomic concept P is a subset of Δ noted P^I
- The extension of the negation of an atomic concept $\neg P$ is Δ - P^I
- The extension of an atomic role R is a subset of $\Delta \times \Delta$ noté R^I
- The extension of a general concept is:
  - $\top^I = \Delta$
  - $\bot^I = \emptyset$
  - $(C \sqcap D)^I = C^I \cap D^I$
  - $(\forall R.C)^I = \{\mathrm{x} \mid \mathrm{x} \in \Delta : \forall y, < x, y > \in R^I \Rightarrow y \in C^I\}$
  - $(\geq n\ \mathrm{R})^I = \{\mathrm{x} \in \Delta : |\{y :< x, y > \in R^I\}| \geq n\}$
  - $(\leq n\ \mathrm{R})^I = \{\mathrm{x} \in \Delta : |\{y :< x, y > \in R^I\}| \leq n\}$

Definition 1 (TC models) *An interpretation I is a model for a terminological component TC iff* $A^I = C^I$ *for every concept definition* $A \equiv C$ *in the terminology, and* $A^I \subseteq C^I$ *for every inclusion* $A \sqsubseteq C$

The subsumption is defined as follows:

Definition 2 (Subsumption) *A concept D subsumes a concept C wrt TC iff for all models I of TC, the extension of C is included in the extension of D:* $C \sqsubseteq D$ *iff* $C^I \subseteq D^I$ *for every model I of TC.*

Example 3
$polygon \equiv closed\_shape \sqcap \geq 1\ has\_side.$
$non\_polygon \equiv closed\_shape \sqcap \leq 0\ has\_side.$
$triangle \equiv three\_sided.$
$three\_sided \equiv polygon \sqcap \geq 3\ has\_side \sqcap \leq 3\ has\_side.$
$four\_sided \equiv polygon \sqcap \geq 4\ has\_side \sqcap \leq 4\ has\_side.$
$parallelogram \equiv four\_sided \sqcap parallel\_sides.$
$rectangle \equiv parallelogram \sqcap square\_angles.$
$square \equiv rectangle \sqcap equal\_sides.$
$circle \equiv non\_polygon \sqcap has\_radius.$
$ellipse \equiv non\_polygon \sqcap has\_two\_focuses.$
$empty\_car \equiv car \sqcap \leq 0\ has\_load.$
$empty\_train \equiv train \sqcap \forall has\_car.empty\_car.$

Classification allows the automatic generation of the taxonomy partially represented in figure 1.

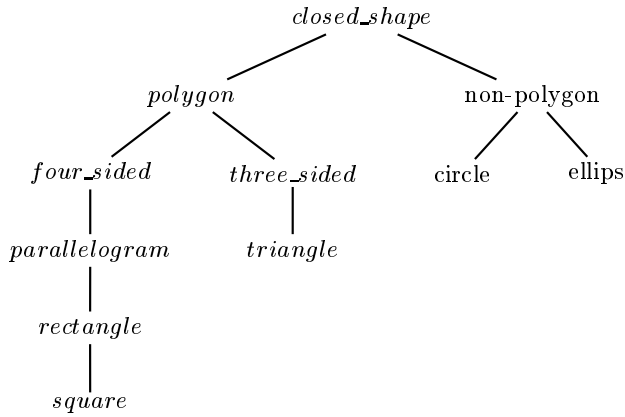


Fig. 1. Partial taxonomy for TC

2.2 Horn Clause Component

A Horn clause component R in CARIN is a set of Horn clauses of the form: $q(\bar{Y}) \leftarrow p_1(\bar{Y}_1) \wedge \ldots \wedge p_m(\bar{Y}_m)$ where $\bar{Y}, \bar{Y}_1, \ldots, \bar{Y}_m$ are tuples of variables or constants. The clauses are safe (i.e., only variables belonging to one or more $\bar{Y}_i$ should be allowed in $\bar{Y}$). The predicates p_i occurring in the body of the clause are of two kinds. They can be either *DL predicates*, (i.e., *concept* or *role* predicates), or *ordinary* predicates. A concept atom for concept c defined in the terminology TC is a literal of the form $c(X)$. A role atom for role r used in the TC terminology is a literal of the form $r(X, Y)$. Since an *ordinary predicate* does not correspond

to any concept or role of TC, it can therefore be of any arity. The predicate of the head of the clause, p, must be an ordinary predicate: CARIN does not allow concept or role atoms to appear in the conclusion of Horn clauses, since it is assumed that TC *completely* describes the terminological background knowledge. Finally, we are only addressing non-recursive Horn clause components, although [LR98b] also considers recursive Horn clause components.

Example 4

$has\_square\_car(X) \leftarrow train(X), has\_car(X,Y), car(Y), square(Y).$
$same\_shape(X,Y) \leftarrow X \neq Y, square(X), square(Y).$
$same\_shape(X,Y) \leftarrow X \neq Y, rectangle(X), rectangle(Y).$
$\vdots$
$pattern(X,Y,Z) \leftarrow train(X), has\_car(X,Y), car(Y), has\_car(X,Z), car(Z),$
$\quad square(Z), has\_load(Z,U), same\_shape(Y,U).$
$east\_train(X) \leftarrow train(X), has\_car(X,Y), car(Y), has\_car(X,Z), car(Z),$
$\quad has\_load(Z,L), same\_shape(Y,L).$
$east\_train(X) \leftarrow train(X), \forall has\_car.polygon(X).$

In this example, the ordinary predicates are has_square_car, same_shape, pattern and east_train. All other predicates are DL predicates.

2.3 Ground Fact Component

The ground fact component is a set of ground atomic facts concerning either DL or ordinary predicates.

Example 5

$\mathcal{A} = \{train(a), has\_car(a,b), car(b), has\_load(b,c), load(c), same\_shape(b,c),$
$has\_car(a,d), car(d), \leq 0 has\_load(d), car(e), has\_car(a,e), \leq 0\ has\_load(e),$
$east\_train(a)\}.$

This ground fact component describes a train going east. A ground fact component represents a particular world or an interpretation in De Raedt's sense (see section 3.1).

2.4 Expressivity of the Language

Considering formulas in TC or R, some formulas can be represented in our language which cannot be represented in Datalog. For instance, if we consider the TC rule $empty\_train \sqsubseteq train \sqcap \forall has\_car.empty\_car$, its translation in First Order Logics is $\forall X empty\_train(X) \rightarrow (train(X) \wedge (\forall Y has\_car(X,Y) \rightarrow empty\_car(Y)))$, which is obviously not a Horn clause because of the universal quantification of "car" variables that occur only in the body of the clause[2].

[2] This type of formula is a restricted case of first order languages with existential quantified variables, which has already been studied for learning, for example in [Gon97, NCVLRDR99].

Moreover, cardinality connectives in the DL allows for smooth and sound handling of simple numeric constraints. Finally, there is also some expressivity gain wrt representation of the ground fact component. In a CARIN$_{\mathcal{ALN}}$ ground fact component, we can express facts concerning both ordinary and DL predicates. As a consequence, we can easily express incomplete information about the observations. For instance, we can represent that a train has more than four cars, without explicitly naming them: $train \sqcap \geq 4 has\_car$.

2.5 Reasoning in CARIN

The key reasoning problem in CARIN is the *existential entailment* problem, fully described in [LR98b] for CARIN-$\mathcal{ALCNR}$, a more complex CARIN language than the one we address here. The existential entailment algorithm standing at the core of several reasoning problems in CARIN, and will be used to order hypothesis our hypothesis space and to check example coverage (see section 4).

Formally, the existential entailment problem for CARIN-$\mathcal{ALN}$ as follows.

Definition 3 (Existential entailment) *Let TC be a terminology in $\mathcal{ALN}$, let β and Q_i be existential sentences of the form: $(\exists \bar{Y})\ p1(\bar{Y_1}) \wedge \ldots \wedge pm(\bar{Y_m})$ where p_i are either roles or concepts occurring in TC, $\bar{Y_i}$ are tuples of variables and constants and $\bar{Y} \subseteq \bar{Y_1} \cup \ldots \cup \bar{Y_m}$, let Q be a sentence of the form $Q_1 \vee \ldots \vee Q_n$. The variables that are not existentially quantified in Q or β are considered universally quantified. Note that any variable appearing in Q must also appear in β.*
The existential entailment problem is to decide whether $\beta \cup TC \models Q$.

In [LR98b], an existential entailment algorithm is given for the language CARIN-$\mathcal{ALCNR}$. This algorithm is based on the construction of *constraint systems* that represent set of models of $\beta \cup TC$.

The algorithm begins with an initial constraint system, $S\beta$, representing the set of all models of $\beta \cup TC$ (e.g. a and b are individuals, the presence of $C(a)$ and $R(a, b)$ in β leads to put $a : C$ and $a\ R\ b$ in $S\beta$). Note that TC is taken into account (e.g. if $C(a)$ is in β and if the sentence: $C \equiv D \sqcap E$ is in TC, them both $a : C$, $a : D$, $a : E$ are in $S\beta$). A set of propagation rules $\mathcal{PR}$ is then applied on $S\beta$ to obtain a set of completions $\mathcal{S}$. A propagation rule corresponds to one of the connectives in the DL and allows us to add some implicit constraints (in this case, it is a kind of saturation) or to make some non-deterministic choices (such propagation rules are called *non-deterministic rules*). Propagation rules concerning the language $\mathcal{ALCNR}$ are fully described in [LR98b], p176.

We will now describe some of these propagation rules for $\mathcal{ALN}$. We say that an individual t is a *R-successor* of an individual s in a constraint system S if $s\ R\ t \in S$.

The propagation rule linked to the concept conjunction connective, denoted $\rightarrow_{\sqcap}$, is as follows (S is the constraint system, s and t are individuals, C_1, C_2 and C are concepts and R is a role):

$S \rightarrow_{\sqcap} S \cup \{s : C_1, s : C_2\}$ if
1. $s : C_1 \sqcap C_2$ belongs to S,
2. $s : C_1$ and $s : S_2$ are not both in S.

The propagation rule linked to value restriction on roles, $\rightarrow_{\forall}$, is:
$S \rightarrow_{\forall} S \cup \{t : C\}$ if
1. $s : \forall R.C$ belongs to S,
2. t is an R-successor of s,
3. $t : C$ is not in S.

$s : \forall R.C$ means that if there exists individuals that are in relation with s by R, these individuals must belong to the concept C. For instance, $s : \forall child.Happy$ expresses the fact that if s has children, all her children are happy. If we know that t is a child of s (s *child* $t \in S$) we must add the fact that t is happy ($t : Happy$ is added to S) according to $s : \forall child.Happy$. The $\rightarrow_{\forall}$ propagation rule leads us to add such constraints.

The following propagation rule, $\rightarrow_{\geq}$, is linked to number restriction on roles:
$S \rightarrow_{\geq} S \cup \{s\ R\ y_i, i \in 1, \ldots, n\} \cup \{y_i \neq y_j, i, j \in 1, \ldots, n, i \neq j\}$ if
1. $s : (\geq nR)$ is in S,
2. $y_1, \ldots, y_n$ are new variables,
3. there are no n pairwise separated R-successors of s in S,
4. s is not blocked (blocked variables are used in order to avoid infinite loops, see [LR98b] for more details)

The obtained completions correspond to different refinements of the constraint system. Some completions represent unsatisfiable constraint systems (e.g. $\{s : A, s : \neg A\}$ states that the individual s belongs both to concept A and to its negation). Clash-free completions represent a subset of the models of $\beta \cup TC$, but the entailment of a formula Q from a clash-free completion can be done by checking whether or not the formula is satisfied in one canonical model of the completion. Finally, for each clash-free completion of $\mathcal{S}$, a canonical interpretation is built and existential entailment checks whether or not Q is satisfied in all the canonical interpretations obtained.

The proof of correctness of the algorithm for $\mathcal{ALCNR}$ has been given in [LR98b] with some complexity results. If we consider $\mathcal{ALN}$ as the DL language, the existential entailment problem is much simpler. An optimized version of the existential entailment algorithm for $\mathcal{ALN}$ is currently being implemented. Note that the $\rightarrow_{\geq}$ rule creates additional variables (or additional constants in our framework) in the constraint system in order to ensure that the $\geq nR$ is satisfied. This is the reason why a Horn clause of the form: $p(X) \leftarrow R(X, Y)$ can be triggered by a ground fact component $\mathcal{A}$ even if no statement of the form $R(a, b)$ occurs in $\mathcal{A}$.

Example 6 *Let* $TC = \{square \equiv rectangle \sqcap equal\_sides;\ rectangle \equiv parallelogram \sqcap square\_angles\}$ *(for simplicity, we assume here that parallelogram, equal_sides and square_angles are atomic concepts).*

$\mathcal{A}$=$\{train(a), \geq 1\ has\_car(a), \forall has\_car.rectangle(a), \forall has\_car.equal\_sides(a)\}$. *The initial constraint system of* $\mathcal{A} \cup \mathcal{TC}$, $S\beta$ *is* $\{a : train, a :\geq\ 1\ has\_car,\ a : \forall\ has\_car.rectangle,\ a : \forall has\_car.equal\_sides,\ a : \forall has\_car.parallelogram,\ a : \forall has\_car.square\_angles\}$. *After applying the propagation rules of* $\mathcal{PR}_{\mathcal{ALN}}$, *the constraint system* S *is* $\{a : train,\ a :\geq 1\ has\_car,\ a : \forall has\_car.rectangle,\ a : \forall has\_car.equal\_sides,\ a : \forall has\_car.parallelogram,\ a : \forall has\_car.square\_angles,$ *a has_car b,* b: *rectangle,* b: *equal_sides,* b: *parallelogram,* b: *square_angles*$\}$

The constant b has been added because of the $\rightarrow_{\geq}$ *rule (and thereafter the constraint a has_car b). The* $\rightarrow_{\forall}$ *rule yields the addition of the constraints* b : *rectangle,* b : *equal_sides,* b : *parallelogram,* b : *square_angles. If* H_1 *is the clause* $p(X) \leftarrow has\_car(X, Y), square\_angles(Y)$, *we obtain that* $H_1, \mathcal{A}, TC \models p(a)$ *since* S *is a model of the antecedent of* H_1 *(with* $Y = b$*).*

Let H_2 *be the clause:* $p(X) \leftarrow has\_car(X, Y), square(Y)$. S *is also a model of the antecedents of* H_2, *but the evaluation of* H_2 *needs to use a special procedure (described p. 195 in [LR98b]) which checks entailment of a ground atom of a concept or role predicate. The evaluation of* H_2 *leads first to state that* $H_2, \mathcal{A}, TC \models p(a)$ *iff* $\mathcal{A}, TC \models \exists Y, has\_car(a, Y), square(Y)$. *Let* $compl(\mathcal{A})$ *be the set of ground facts constructed for* S *(e.g. the presence of the constraint* b : *rectangle in* S *leads the addition of the ground fact* $rectangle(b)$ *in* $compl(\mathcal{A})$*).* $square(b)$ *is not in* $compl(\mathcal{A})$, *but the special procedure checks entailment of* $square(b)$ *according to the following rule: if* $C(s)$ *is an atom, where* C *is a concept name defined in* TC *by the description* D *(i.e.,* $C \equiv D$ *is in* TC*),* $compl(\mathcal{A}) \models C(s)$ *if* $D = C_1 \sqcap C_2$ *and* $compl(\mathcal{A}) \models C_1(s)$ *and* $compl(\mathcal{A}) \models C_2(s)$. *In our example,* $compl(\mathcal{A}) \models square(b)$ *since* $square \equiv rectangle \sqcap equal\_sides$ *and* $compl(\mathcal{A}) \models rectangle(b)$ *and* $compl(\mathcal{A}) \models equal\_sides(b)$.

Note that $\mathcal{A}$ *entails the antecedents of* H_1 *and* H_2 *without them being all explicit in the initial* $\mathcal{A}$.

This example illustrates that traditional Horn clause inference mechanisms are inadequate to reason in CARIN knowledge bases. Another aspect concerns the fact that a CARIN KB may entail the disjunction of the antecedents of two clauses without entailing either of them (i.e., we cannot consider each clause in isolation as it is the case in the traditional ILP approaches).

The existential entailment provides the basis for a sound and complete reasoning algorithm for CARIN-$\mathcal{ALN}$ knowledge bases. For learning tasks, we use particular cases of the existential entailment, which are described in sections 4 and 5. The complexity results about these tasks are given in section 6.

3 The Learning Setting

Our learning framework is that of learning from interpretations [DR97]. Learning from interpretations upgrades the attribute value setting in that it allows a system to learn non recursive First Order Logic programs from ground sets of facts (interpretations). The implicit assumption is that each example is standalone, (i.e. it contains all necessary information for the learning task to perform)

and that the coverage test of a hypothesis wrt an example can be extensional (local) yielding a coverage test wrt to a set of examples linear in the number of examples.

Let us focus here on the reformulation of this setting for a classification learning task, as presented in [BDRJ99]. Given :

- a set C of classes (each class is labelled by a predicate p_i, $1 \leq i \leq n$, all class predicates with the same arity)
- a set E of classified examples (each example is of the form $(e, p_i(\bar{a_j}))$, where p_i is the class of the example and e is a set of ground facts where constants $\bar{a_j}$ occur
- a background knowledge BK

Find a hypothesis H such that for all $(e, p_i(\bar{a_j}) \in E$

- $H \wedge e \wedge BK \models p_i(\bar{a_j})$, and
- $H \wedge e \wedge BK \not\models p_j(\bar{a_j})$, for $1 \leq j \leq n$ and $j \neq i$.

In this setting, and after the presentation of CARIN-$\mathcal{ALN}$, let us now specify now what are the languages of our examples, background knowledge and hypotheses.

3.1 Background Knowledge

Our background knowldedge is represented as a CARIN-$\mathcal{ALN}$ knowledge base. The hierarchical part of the background knowledge is represented as a CARIN-$\mathcal{ALN}$ terminological component TC (see section 2.1), and any knowledge that cannot be represented in the TC language is represented in R, the Horn clause component of CARIN-$\mathcal{ALN}$. This includes, for instance, clauses that require existential variables, clauses that express equalities between role restrictions, or rules that express relationships among more than two objects (see example 4).

3.2 Examples

An example e_i is represented in a ground fact component A_i. A ground fact component represents exactly one example. The set of classified examples is therefore a set of ground fact components representing the classified examples. Each example is a model of the set of clauses defining its class predicate. In addition, each example e_i is associated to a tuple of constants $\bar{a_i}$ representing the instanciation of the target predicate for e_i ($\bar{a_i}$ is called the target tuple of constants of e_i).

Example 7 *east_train*(X) *is the target predicate,* e_1 *is represented by* $\mathcal{A}_1 = \{train(a), has\_car(a,b), car(b), square(b), has\_load(b,c), load(c), has\_car(a,d), car(d), ellipse(d), has\_load(d,e), load(e), \leq 2\ has\_car(a), \leq 1\ has\_load(b), \leq 1\ has\_load(d), same\_shape(c,e)\}$*, we associate to* e_1 $\bar{a_1} = a$.

3.3 Hypothesis Language

A hypothesis is a set of Horn clauses described using the language of the Horn clause component and concluding on the target concept. This means that the learning task we address is not to extend TC, but to extend R with new concept definitions. TC is assumed to be static in this work.

Example 8
$H = \{east\_train(X) \leftarrow train(X), \geq 1\ has\_car(X), \forall\ has\_car. \leq 1\ has\_load(X), has\_car(X,Y), polygon(Y); east\_train(X) \leftarrow train(X), \leq 0\ has\_car(X)\}$

4 Testing Coverage of Examples

We now address the problem of testing whether an example e_i, represented as a CARIN-$\mathcal{ALN}$ ground fact component, is covered by a hypothesis H defining a target concept p, wrt a CARIN-$\mathcal{ALN}$ background knowledge BK that includes a terminological component and a Horn rule component. This is the case if $p(\bar{a})$, where $\bar{a}$ is the tuple of constants associated to e_i, is logically entailed by BK. More formally:

Definition 4 (Covers relation) *Given a CARIN-$\mathcal{ALN}$ knowledge base BK (BK contains a terminological component TC and a Horn rule component R), a hypothesis H defining a target concept p, e_i an example represented by the ground fact component $\mathcal{A}_i$ together with $\bar{a}_i$ its target tuple of constants, H covers e_i if and only if:*

$$BK \cup H \cup \mathcal{A}_i \models p(\bar{a}_i)$$

Testing coverage amounts to computing the canonical model(s) of $TC \cup R \cup H \cup \mathcal{A}_i$ and then to checking whether $p(\bar{a}_i)$ is satisfied in such model(s).

Existential entailment (see section 2.5), in the case in which β is a ground fact component $\mathcal{A}_i$, a conjunction of ground facts representing an example, enables us to decide whether

$$\mathcal{A}_i, TC \models Q_1 \vee \ldots \vee Q_n$$

i.e., to decide whether the disjunction of a set of Horn clause bodies is entailed from $\mathcal{A}$ and TC[3].

We therefore need to reformulate the coverage problem as one that can be handled by existential entailment. One possibility for compute these canonical models is to handle first the background knowledge expressed in TC as well as the logical rules defining the DL connectors and their interactions (it is done using the constraint system approach described in section 2.5), yielding a completed ground fact component $compl(\mathcal{A}_i)$. The second step is then to compute

[3] It has been shown in [LR98b] that entailing antecedents of CARIN rules is not complicated by the fact that they contain ordinary predicates in addition to concepts and roles

the models of $compl(\mathcal{A}_i) \cup R \cup H$. This can be done by a forward chaining (saturation)[4] of $compl(\mathcal{A}_i)$ given $\mathcal{R} \cup H$. Since the propagation and forward chaining algorithms[5] are correct and complete, we will obtain *every* logical consequences of $BK \cup H \cup \mathcal{A}_i$.

However, we just need to obtain logical consequences about $p(\bar{a_i})$. This is why we present in the next section a coverage computation guided by the target predicate, which can avoid to consider every rule of R. One method would be to try to prove $p(\bar{a_i})$ using clauses of $R \cup H$ and facts of $compl(\mathcal{A}_i)$. The coverage testing of a set of n examples therefore leads us to unfold rules for each example (i.e., we will unfold $p(\bar{a_1}), \ldots, p(\bar{a_n})$). Another approach consists of applying first an unfolding process on $p(\bar{X})$ wrt clauses of $R \cup H$ independently of any example. This process generates the disjunction of all possible ways to prove $p(\bar{X})$ given $R \cup H$. We can see this step as a kind of pre-compilation of $\mathcal{R} \cup H$ wrt $p(\bar{X})$. This pre-compilation is made once for each target predicate definition, and it is then sufficient to compute for an example its completions wrt $\mathcal{TC}$ and to check whether these completions satisfy any of the disjuncts obtained in the previous step.

Re-using the arguments that took place some years ago in the Explanation based Learning community, we cannot state that one approach is better than the other independently of a particular CARIN KB and in particular independently of any example presentation : in some cases the pre-compilation will cost a lot with few advantages, in other cases the pre-compilation may save of lot of computation time. In this paper, we choose to present the algorithm based on pre-compilation but the precise comparison of the two approaches is outside the scope of this paper.

4.1 Coverage Test Computation

The following coverage test algorithm uses algorithms described in [LR98b].

$coverage\_test(H, \{\mathcal{A}_1, \ldots, \mathcal{A}_n\}, TC, R)$

1. Precompilation of $R \cup H$ wrt p: build the disjunction $Q_1 \vee \ldots \vee Q_n$ such that for all $\mathcal{A}_i$,
$$\mathcal{A}_i, TC \models Q_1 \vee \ldots \vee Q_n \text{ iff } BK, H, \mathcal{A}_i \models p(\bar{a_i})$$
2. Coverage test: for i $\in$ {1,...,n}, use the existential entailment algorithm in order to decide whether $\mathcal{A}_i, TC \models Q_1 \vee \ldots \vee Q_n$ or not.

More precisely, the first step amounts to partial evaluation [VHB88] of $p(\bar{X})$ wrt clauses of $R \cup H$, i.e., to compute (not guided by any example) all the ways to

[4] This is not a simple forward chaining since the evaluation of concept and role predicates requires the application of a special procedure (see example 6). In the remainder of the paper by forward chaining we intend forward chaining upgraded with this special evaluation procedure.

[5] Classification systems such as CLASSIC are usually implemented as forward-chaining inference systems.

prove $p(\bar{X})$. As partial evaluation is performed wrt $R \cup H$ and since CARIN does not allow concept or role atoms to appear in the conclusion of R clauses, only ordinary predicates are "unfolded". As $R \cup H$ is non recursive, this process terminates, yielding a disjunction of Q_i. The DL part of Horn clauses obtained is left unchanged by this step : the saturation of $\mathcal{A}_i$ wrt TC is done in the next step, as part of the existential entailment algorithm.

The goal of the second step is to check whether:
$\mathcal{A}_i, TC \models Q_1 \vee \ldots \vee Q_n$ where the Q_is are the disjuncts obtained by the partial evaluation process.

Let $\mathcal{ADL}$ be the set of ground facts in $\mathcal{A}_i$ built on role and concept predicates and let $\mathcal{AR}$ be the set of ground facts in $\mathcal{A}_i$ built on ordinary predicates. The set of propagation rules $PR_{\mathcal{ALN}}$ is applied on the initial constraint system of $\mathcal{ADL}$. The obtained completions correspond to the set of models of $\mathcal{ADL} \cup TC$. Canonical models are computed and existential entailment checks whether one of the disjuncts obtained during at step 1 is satisfied in all the canonical models obtained. If it is the case, the hypothesis H covers the example represented by $\mathcal{A}_i$, else H does not cover the example.

Example 9 *Let us consider H after example 8 and e_1 after exemple 7, let BK be TC be described after example 3 and let R be empty, H_1 covers e_1 since $TC \cup H \cup \mathcal{A}_1 \models east\_train(a)$. If we consider the ground fact component of example 6, $TC \cup H \cup \mathcal{A} \not\models east\_train(a)$ since $TC \cup H \cup \mathcal{A} \not\models \forall has\_car. \leq 1\ has\_load(a)$.*

Example 10 *Consider e as $\mathcal{A} = \{train(a), \geq 2\ has\_car(a), \forall has\_car.car(a), \forall has\_car.square(a)\}$. If $H = \{east\_train(X) \leftarrow train(X), \geq 1\ has\_car(X), \forall has\_car.polygon(X)$, H covers e wrt TC and R (empty).*
If we now consider R as in exemple 4 and $H = \{east\_train(X) \leftarrow train(X), \geq 1\ has\_car(X), has\_car(X,Y), car(Y), has\_car(X,Z), car(Z), same\_shape(Y,Z)$, H covers e wrt TC and R.

Another advantage in using the existential entailment algorithm instead of a simple Horn clause inference mechanism is that it allows one to detect when the *disjunction* of Q_is is entailed by $TC \cup H \cup \mathcal{A}_i$, while none of the Q_is are true in isolation (see [LR98b], p172 for a detailed explanation of using a forward chaining mechanism).

Example 11 *Consider e as $\mathcal{A} = \{train(a), has\_car(a,b), square(b)\}$ and $H = \{east\_train(X) \leftarrow train(X), has\_car(X,Y), rectangle(Y), \geq 2\ has\_load(Y)(H_1)$; $east\_train(X) \leftarrow train(X), has\_car(X,Y), equal\_sides(Y), \leq 4\ has\_load(Y)(H_2)\}$. H covers e wrt TC and R (empty) since $body(H_1) \vee body(H_2)$ is entailed while neither $body(H_1)$ nor $body(H_2)$ are entailed (no information about the number of loads of b is present in e).*

5 Hypothesis Ordering

Lastly, let us define the partial ordering that structures our search space. Roughly speaking, it is an extension of generalized subsumption [Bun88] when background

knowledge is made of a set of non recursive horn clauses plus a terminological component.

A hypothesis is a set of Horn clauses described using the language of the Horn clause component of CARIN-$\mathcal{ALN}$ and concluding on the target concept. Intuitively, a hypothesis H_g is more general than a hypothesis H_s if it covers more examples. For simplicity, we only consider here conjunctive hypotheses. If $head(H_s) = head(H_g)$[6], as for generalized subsumption, we only need to compare hypotheses bodies (i.e., existentially quantified conjunctions of literals). More formally:

Definition 5 (Hypothesis order) *Let H_g and H_s be two hypotheses of the form:*

$$H_g = p(\bar{X}) \leftarrow p_1(\bar{Y_1}) \wedge \ldots \wedge p_m(\bar{Y_m})$$
$$H_s = p(\bar{X}) \leftarrow p'_1(\bar{Y'_1}) \wedge \ldots \wedge p_n(\bar{Y_n})$$

H_g is more general than H_s wrt TC and R iff the body of H_s, together with TC and R, logically entails the body of H_g:

$$TC, R, (\exists \bar{Y'}) p'_1(\bar{Y'_1}) \wedge \ldots \wedge p_n(\bar{Y_n}) \models (\exists \bar{Y}) p_1(\bar{Y_1}) \wedge \ldots \wedge p_m(\bar{Y_m})$$

Once again, we can use a particular case of the existential entailment in order to determine whether a hypothesis is more general than another hypothesis. Let us recall that existential entailment decides whether $\beta \cup TC \models Q_1 \vee \ldots \vee Q_n$ where β and Q_i are existential sentences of the form: $(\exists \bar{Y}) p_1(\bar{Y_1}) \wedge \ldots \wedge p_m(\bar{Y_m})$. In the case where $n = 1$, the existential entailment problem enables us to decide whether $\beta \cup TC \models Q_1$ (this subcase is refered to as *query containment* in [LR98b]).

Definition 5 states that we need to compare the bodies of H_s and H_g, denoted $body(H_s)$ and $body(H_g)$[7], wrt background knowledge expressed as $TC \cup R$. The existential entailment algorithm enables us to decide whether $body(H_s) \cup TC \models body(H_g)$, i.e., to compare the two hypotheses wrt TC but not wrt R.

Let us recall that the first step of the existential entailment algorithm consists of computing the completions wrt TC of the left hand formula of the implication (a ground fact component in the coverage test, an existential sentence in the subsumption ordering problem, but the principle of the algorithm is similar). Let us call $compl(H_s)$ the existential sentence corresponding to one completion of $body(H_s)$ obtained as the application of the set of propagation rules $\mathcal{PR}_{\mathcal{ALN}}$ to $body(H_s)$. $compl(H_s)$ is, so to say, saturated according to TC.

It is now sufficient to apply forward chaining wrt R on $compl(H_s)$ and to check whether the obtained sentence entails $body(H_g)$. H_g is more general than H_s if this is the case, otherwise not. Once again, the completeness and soundness of existential entailment (proved in [LR98b]) and of forward chaining enables us to have a sound and complete inference mechanism for checking subsumption between two hypotheses.

[6] We assume here that all hypotheses of the target concept are rectified, (i.e., they have the same head variables).

[7] $body(H_g)$ and $body(H_s)$ are existential sentences (the presence of ordinary predicates is not a problem).

Example 12 *Let TC be reduced to the sentence :*
$circle \equiv non\_polygon \sqcap has\_radius$
Let R be reduced to the Horn clause :
$non\_polytrain(X) \leftarrow train(X), has\_car(X,Y), non\_polygon(Y).$
Let H_s *be:* $east\_train(X) \leftarrow train(X), \geq 1 has\_car(X), \forall has\_car.circle(X).$
Let H_g *be:* $east\_train(X) \leftarrow non\_polytrain(X).$

H_g *is more general than* H_s *since:*
$train(X), \geq 1 has\_car(X), \forall has\_car.circle(X) \cup TC \cup R \models non\_polytrain(X).$

6 Complexity Study

It has been shown in [LR98b] that the time complexity of existential entailment is deterministic doubly exponential since this problem requires that all completions be checked, and there is, in the worst case, a doubly exponential number of completions. However, when the terminological language is $\mathcal{ALN}$ and not $\mathcal{ALCNR}$, the existential entailment problem is much simpler and can be optimized. For the problem of query expansion in CARIN-$\mathcal{ALN}$, which is a similar but slightly more complex problem than our coverage or subsumption test[8], [GLMC] exhibits a time complexity exponential in the depth of ordinary atoms unfolding and in the maximal size of concept expressions in the query. This shows that unless the Horn clause component is restricted, the complexity of both the coverage test and the subsumption checking in CARIN-$\mathcal{ALN}$ are dominated by the the complexity of the coverage test and the subsumption checking in Horn clause logic. This rough complexity study has to be further refined in the case of restricted Horn clause components (ij-determinate [MF90] or k-local [KL94]).

7 Related Work and Future Directions

We have presented a coverage test and a subsumption ordering for learning from interpretations in a hybrid language that combines a simple description logic with horn clause logic.

LIFE ([AKP91]) is a language combining logic programming with ψ-terms used to represent sub-typing in record-like data structures. On the one hand, the variables in ψ-terms enable one to express co-reference constraints which cannot be expressed using $\mathcal{ALN}$. On the other hand, ψ-terms only allow attributes (i.e., functional roles), they are therefore not suited to express, for instance, number restrictions about role fillers. Note that adding the *same-as* constructor of CLASSIC would allow us to express limited co-reference constraints, however Cohen and Hirsh showed in [CH92] that the DL with only the $\sqcap$, $\forall$ and *same-as* constructors is not pac-learnable.

[8] Roughly speaking, query expansion amounts to computing all possible rewritings of a given query, given a terminological component TC and a horn clause component R.

The field of Description Logics has raised increasing interest in the last few years in the Machine learning community. [KM94] presents a constructive induction program named KLUSTER that uses a DL formalism to represent concept definitions. This DL is provided with a *Least Common Subsumer* algorithm. KLUSTER upgrades a T-Box given examples represented as A-Boxes, but it does not learn in a hybrid language target concept language as we do, and does not handle either hybrid background knowledge, including a horn clause component.

Cohen and Hirsh [CH94b, CH94a] give theoretical and experimental results on the learnability of description logics. In particular, they prove that C-CLASSIC is PAC-learnable. From a practical point of view, the authors propose several algorithms that allow C-CLASSIC concepts to be learned from positive and negative examples of these concepts. The language of both concepts and examples is the terminological language of the DL. Note that this single trick representation leads to a loss of information since the examples are abstracted. For instance, let the example be defined by the initial description: $train(a), has\_car(a,b), circle(b)$,
$has\_car(a,c), rectangle(c)$, the abstraction of the example in C-CLASSIC is $(train \sqcap \geq 2\ has\_car)$. In our approach, examples are represented in a ground fact component, which enables us to reason using the whole information.

In [CH94b, CH94a], the *covers* relation that specifies how hypotheses relate to examples is the subsumption relation: a hypothesis H covers an exemple e if and only if $e \sqsubseteq H$ (i.e., H subsumes e). The aim is then to find a hypothesis H that covers all positive examples (completeness) and none of the negative examples (consistency). As C-CLASSIC only contains a limited kind of disjunction (the ONE-OF connective), many target concepts of practical interest can not be expressed using a single term of C-CLASSIC. One way to overcome this limitation is to consider algorithms that learn a disjunction of terms rather than a single term. A hypothesis $H \equiv H_1 \vee H_2 \ldots \vee H_n$ then covers an example e (represented as a concept) if and only if $\exists H_i \in H$, $e \sqsubseteq H_i$ (i.e., H_i subsumes e).

However, in [FP96], Pitt and Frazier shows that the union of terms of C-CLASSIC can subsume a concept even though neither member of the union susbsumes this concept (let us recall that there is no closed-world assumption in DLs). Reasoning about the possibility of such interactions makes the problem of learning in this situation challenging. In our framework, the existential entailment enables us to take into account such interaction since clauses are not considered in isolation (see [LR98b]). Finally, Cohen and Hirsh suggest that learning systems based on description logics may prove to be a useful complement to ILP systems. The work presented here is a first tentative at combining the two frameworks.

The perspectives of this work are numerous. Now that the framework in CARIN-$\mathcal{ALN}$ has been set and the complexity results seems encouraging, an actual learning system can be developed, providing a solid base for experimentation. A lgg-based approach does seem, intuitively, the most appropriate direc-

tion to investigate, as input horn clauses to the lgg process, once completed by the different inference mechanisms described in the paper will be quite large. A more promising direction will be to design a data-driven top-down operator for CARIN-$\mathcal{ALN}$, as in [AR00] or [BCS98]. A more theoretical line of research will be to look for a (if any) restricted CARIN language that may be PAC-learnable.

Acknowledgements We wish to thank M.-Ch. Rousset, for the many discussions we had concerning the CARIN language and the existential entailment algorithm, as well as J-U Kietz for helpful discussions on previous versions of the paper. Last but not least, we are grateful to Lise Fontaine, who proof-read a nearly final version of the paper.

References

[AKP91] H. Ait-Kaci and A. Podelski. Towards the meaning of LIFE. In J. Maluszynski and M. Wirsing, editors, *Proceedings on 3rd International Symposium on Programming Language Implementation and Logic Programming*, pages 255–274, Berlin, 1991.

[AR00] E. Alphonse and C. Rouveirol. Lazy propositionalisation for relational learning. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, Berlin, 2000. IOS Press. to appear.

[BCS98] P. Brézellec, M. Champesme, and H. Soldano. Tabata, a learning algorithm searching a minimal space using a tabu strategy. In H. Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence*, pages 420–424, Brighton, 1998. Wiley.

[BDRJ99] H. Blockheel, L. De Raedt, and B. Jacobs, N.and Demoen. Scaling up inductive logic programming by learning from interpretaions. *Data Mining and Knowledge Discovery*, 3(1):59–93, 1999.

[BPS94] A. Borgida and P. F. Patel-Schneider. Complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, 1:278–308, 1994.

[Bra78] R.J. Brachman. A structural paradigm for representing knowledge. Technical Report 3605, BBN Report, 1978.

[Bun88] W. Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[CH92] W. W. Cohen and H. Hirsh. Learnability of the classic knowledge representation language. Technical report, ATT Bell Laboratories, New York, 1992.

[CH94a] W. W. Cohen and H. Hirsh. The learnability of the description logics with equality constraints. *Machine Learning*, 2(4):169–199, 1994.

[CH94b] W. W. Cohen and H. Hirsh. Learning the CLASSIC description logic: Theoretical and experimental results. In *International Conference on Knowledge Representation and Reasoning*, pages 121–133. 1994.

[DLNN91] F.M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J.A. Allen R. Fikes and E Sandewall, editors, *Principles of Knowledge Representation and Reasoning: 2nd International Conference*, pages 151–162, Cambridge, Mass., 1991.

[DLNS91] F.M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. A hybrid system with datalog and concept languages. In E. Ardizzone, S. Gablio, and F. Sorbello, editors, *Trends in Artificial Intelligence*, volume 549 of *Lecture Notes in Artificial Intelligence*, pages 88–97. Springer-Verlag, 1991.

[DR97] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.

[FP96] M. Frazier and L. Pitt. Classic learning. *Machine Learning*, 25:151–193, 1996.

[GLMC] F. Goasdoué, V. Lattès, and Rousset M.-Ch. The use of carin language and algorithms for information integration: The picsel system. *International Journal of Cooperative Systems*. submitted.

[Gon97] M-E. Goncalves. Handling quantifiers in ILP. In S. Muggleton, editor, *Proc. of the 6th International Workshop on Inductive Logic Programming*, pages 337–357. Springer Verlag, 1997.

[KL94] J-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In William W. Cohen and Haym Hirsh, editors, *Proc. 11th International Conference on Machine Learning*, pages 130–138. Morgan Kaufmann, 1994.

[KM94] J. U. Kietz and K. Morik. A polynomial approach to the constructive induction of structural knowledge. *Machine Learning*, 14(2):193–217, 1994.

[LR98a] A. Levy and M.-Ch. Rousset. Verification of knowledge bases based on containment checking. *Artificial Intelligence*, 101(1-2), 1998.

[LR98b] A. Y. Levy and M.-Ch. Rousset. Combining horn rules and description logics in carin. *Artificial Intelligence*, 104:165–209, 1998.

[MF90] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsma, Tokyo, Japan, 1990.

[MMPS94] D. Michie, S. Muggleton, D. Page, and A. Srinivasan. To the international computing community: A new East-West challenge. Technical report, Oxford University Computing laboratory, Oxford,UK, 1994. URL: ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/trains.tar.Z.

[NCVLRDR99] S.-H. Nienhuys-Cheng, W. Van Laer, J. Ramon, and L. De Raedt. Generalizing refinement operators to learn prenex conjunctive normal forms. In S Džeroski and P. Flach, editors, *Proc. of the 9th International Conference on Inductive Logic Programming*, pages 245–256. Springer Verlag, 1999.

[QDBK96] J. Quantz, G. Dunker, F. Bergmann, and I. Kellner. The flex system. Technical report, KIT-Report, Technische Universität, Berlin, Germany, 1996.

[Val84] L.G. Valiant. A theory of the learnable. *Communications of the ACM*, 27:1134–1142, 1984.

[VHB88] F. Van Harmelen and A. Bundy. Explanation based generalisation = partial evaluation. *Artificial Intelligence*, 36:401–412, 1988.

[WWB+93] J.R. Wright, E.S. Weixelbaum, K. Brown, G.T. Vesonder, S.R. Palmer, J.I. Berman, and H.H. Moore. A knowledge-based configurator that supports sales, engineering and manufacturing at att bell network systems. In *Proceedings of the Innovative Applications of Artificial Intelligence Conferences*, pages 183–193, Menlo Park, California, 1993.

Inverse Entailment in Nonmonotonic Logic Programs

Chiaki Sakama

Department of Computer and Communication Sciences
Wakayama University
Sakaedani, Wakayama 640 8510, Japan
sakama@sys.wakayama-u.ac.jp
http://www.sys.wakayama-u.ac.jp/∼sakama

Abstract. Inverse entailment (IE) is known as a technique for finding inductive hypotheses in Horn theories. When a background theory is nonmonotonic, however, IE is not applicable in its present form. The purpose of this paper is extending the IE technique to nonmonotonic inductive logic programming (ILP). To this end, we first establish a new entailment theorem in normal logic programs, then introduce the notion of contrapositive programs. Finally, a theory of IE in nonmonotonic ILP is constructed.

1 Introduction

Inverse entailment (IE) [11] is one of the basic techniques in inductive logic programming (ILP), which is used for computing inductive hypotheses in the following manner. Given a background Horn logic program B and an example E as a Horn clause, suppose a hypothetical Horn clause H satisfying

$$B \wedge H \models E.$$

By inverting the entailment relation it becomes

$$B \wedge \neg E \models \neg H.$$

Put $\neg\mathsf{Bot}$ as the conjunction of ground literals which are true in every model of $B \wedge \neg E$. Consider the case of $\neg\mathsf{Bot} \models \neg H$.[1] Then, it becomes $H \models \mathsf{Bot}$. Such Bot is effective for reducing the hypothesis space, since a possible hypothesis H is constructed as a clause which subsumes Bot.

A Horn logic program is *monotonic* in the sense that adding a clause to the program never leads to the loss of any conclusions previously proved in the program. However, it is known that Horn logic programs are not sufficiently expressive for the representation of *incomplete knowledge*. In the real world, humans perform *commonsense reasoning* when one's knowledge is incomplete.

[1] Note that $B \wedge \neg E \models \neg H$ does not imply $\neg\mathsf{Bot} \models \neg H$. This is the reason for the incompleteness of IE [15].

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 209–224, 2000.

Commonsense reasoning is *nonmonotonic* in its feature, that is, previously concluded facts might be withdrawn by the introduction of new information. A logic program which has the nonmonotonic feature is called a *nonmonotonic logic program*.

When a background theory is a nonmonotonic logic program, however, the above IE relation does not hold in general. The IE relation is derived from the original entailment as follows:

$$B \wedge H \models E \;\Leftrightarrow\; B \models (H \rightarrow E) \quad (1)$$
$$\Leftrightarrow\; B \models (\neg E \rightarrow \neg H) \quad (2)$$
$$\Leftrightarrow\; B \wedge \neg E \models \neg H. \quad (3)$$

When B is nonmonotonic, there are two problems in the above derivation process. First, in nonmonotonic logic the deduction theorem does not hold in general [14], hence the equivalence relations (1) and (3) do not hold. Second, in nonmonotonic logic programs, a rule presents a derivation rule in one-way direction, so that the contrapositive implication (2) is undefined. These facts mean that the present IE technique cannot be used when a background theory is nonmonotonic. Then, reconstruction of the framework is necessary to apply IE to nonmonotonic ILP.

This paper studies a theory of inverse entailment in *nonmonotonic ILP*. As nonmonotonic logic programs, we consider *normal logic programs*, i.e., logic programs with *negation as failure*. Negation as failure represents default negation in a program, which makes the semantics of programs different from classical logic. We first review the problem of the deduction theorem in nonmonotonic theories, then introduce the new entailment theorem in normal logic programs. Next, we introduce contrapositive rules in normal logic programs and present their semantic properties. Finally, we construct a theory of inverse entailment in normal logic programs.

The rest of this paper is organized as follows. Section 2 presents a theoretical framework used in this paper. Section 3 establishes the new entailment theorem in normal logic programs. Section 4 introduces a framework of contrapositive programs. Section 5 constructs a theory of inverse entailment in normal logic programs, and Section 6 provides examples. Section 7 discusses related issues and Section 8 concludes the paper.

2 Normal Logic Programs

Definitions of basic notions such as *constants*, *functions*, *variables*, *predicates*, and *atoms* follow from those standard in logic programming [9]. A logic program considered in this paper is a *normal logic program* [9,12], which is a set of rules of the form:

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n \quad (4)$$

where each A_i $(0 \leq i \leq n)$ is an atom and *not* presents *negation as failure* (NAF). The atom A_0 is the *head*, and the conjunction $A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$

is the *body* which is identified with the set $\{ A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \}$. We allow a rule with an empty head of the form:

$$\leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{5}$$

which is also called an *integrity constraint*. A rule with an empty body $A \leftarrow$ is identified with the atom A. Throughout the paper a program means a normal logic program unless stated otherwise. A program P is *Horn* if no rule in P contains NAF. The *Herbrand base* $\mathcal{HB}$ of a program P is the set of all ground atoms in the language of P. Given the Herbrand base $\mathcal{HB}$, we define $\mathcal{HB}^+ = \mathcal{HB} \cup \{ not\ A \mid A \in \mathcal{HB} \}$. Any element in $\mathcal{HB}^+$ is called an *LP-literal*. A program, a rule, or an LP-literal is *ground* if it contains no variable. Any variable in a program is interpreted as a free variable. In this case, a program P is semantically identified with its ground instantiation, i.e., the set of ground rules obtained from P by substituting variables in P with elements of the Herbrand universe in every possible way.

An *interpretation* is a subset of the Herbrand base of a program. An interpretation I *satisfies* the conjunction of ground LP-literals $C = A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ if $\{A_1, \ldots, A_m\} \subseteq I$ and $\{A_{m+1}, \ldots, A_n\} \cap I = \emptyset$ (written as $I \models C$). An interpretation I satisfies the ground rule R of the form (4) if $I \models A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$ implies $A_0 \in I$ (written as $I \models R$). In particular, I satisfies the ground integrity constraint of the form (5) if either $\{A_1, \ldots, A_m\} \setminus I \neq \emptyset$ or $\{A_{m+1}, \ldots, A_n\} \cap I \neq \emptyset$. When a rule R contains variables, $I \models R$ means that I satisfies every ground instance of R. When I does not satisfy R (i.e., $I \not\models R$), it is also written as $I \models not\ R$. We define $I^+ \models R$ if $I \models R$ where $I^+ = I \cup \{ not\ A \mid A \in \mathcal{HB} \setminus I \}$. An interpretation which satisfies every rule in a program is a *model* of the program. A model M of a program P is *minimal* if there is no model N of P such that $N \subset M$.

A Horn logic program has at most one minimal model, i.e., the *least model*, which provides the declarative meaning of a program. On the other hand, a normal logic program generally has multiple minimal models. For instance, the program

$$a \leftarrow not\ b$$

has two minimal models $\{a\}$ and $\{b\}$. However, in this program the rule asserts that a is true if b is not true. Since there is no evidence of b, it is natural to conclude a in the program. In this regard, the minimal model $\{a\}$ should be taken as the intended meaning of the program. Thus, in normal logic programs it is necessary to identify the class of minimal models which provide the intended meaning of a program. The *stable model semantics* proposed by Gelfond and Lifschitz [6] is one of the most popular semantics of normal logic programs. A stable model provides a natural meaning for many normal logic programs and coincides with the least model in Horn logic programs. Moreover, the stable model semantics plays an important role to relate logic programming and nonmonotonic formalisms in AI [2].

A formal definition of the stable model semantics is as follows. Let R be a ground rule of the form (4) or (5) and I an interpretation. Then, define R^I as

$$R^I = \begin{cases} A_0 \leftarrow A_1, \ldots, A_m & \text{if } \{A_{m+1}, \ldots, A_n\} \cap I = \emptyset. \\ \mathit{true}, & \text{otherwise.} \end{cases}$$

Given a program P and an interpretation I, the ground Horn program P^I is defined as

$$P^I = \{\, R^I \mid R \text{ is a rule in the ground instantiation of } P \,\}.$$

If the least model of P^I is identical to I, I is called a *stable model* of P.

The intuitive meaning of the transformation from P to P^I is as follows. First assume an interpretation I as a possible set of beliefs. Then, for any ground rule $R:\ A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\, A_{m+1}, \ldots, \mathit{not}\, A_n$, if $\{A_{m+1}, \ldots, A_n\} \cap I = \emptyset$, then the conjunction $\mathit{not}\, A_{m+1}, \ldots, \mathit{not}\, A_n$ is true wrt I and R is simplified as R^I in P^I. Otherwise, R contains some $\mathit{not}\, A_i$ $(m+1 \leq i \leq n)$ such that $A_i \in I$, then I does not satisfy the condition of R and such R is useless for derivation thereby deleted (or replaced by true) in P^I. If the least model of P^I coincides with I, the belief set I is justified and is called a stable model.

A program may have none, one, or multiple stable models in general. A program having exactly one stable model is called *categorical* [2].[2] A stable model is a minimal model and it coincides with the least model in a Horn logic program. A program is *consistent* (under the stable model semantics) if it has a stable model; otherwise a program is *inconsistent*. Any program is assumed to be consistent in this paper.

Example 2.1. The program

$$a \leftarrow \mathit{not}\, b,$$
$$b \leftarrow \mathit{not}\, c$$

has the unique stable model $\{b\}$. Hence, the program is categorical.

The program

$$a \leftarrow \mathit{not}\, b,$$
$$b \leftarrow \mathit{not}\, a$$

has two stable models $\{a\}$ and $\{b\}$, which represents two alternative beliefs.

On the other hand, the program

$$a \leftarrow \mathit{not}\, a$$

has no stable model, hence the program is inconsistent.

If every stable model of a program P satisfies a rule R, it is written as $P \models_s R$. Else if no stable model of a program P satisfies a rule R, it is written as $P \models_s \mathit{not}\, R$. In particular, $P \models_s A$ if a ground atom A is true in every stable model of P; and $P \models_s \mathit{not}\, A$ if A is false in every stable model of P. In contrast, if every model of P satisfies R, it is written as $P \models R$. Note that when P is Horn, the meaning of $\models$ coincides with the classical entailment.

[2] An example of the class of programs with this property is a *locally stratified program*.

3 Entailment Theorem

In classical first-order logic, the following deduction theorem holds.

Given a set of formulas T and a closed formula F,
$T \wedge F \models G$ iff $T \models F \rightarrow G$ for any formula G.

In the context of *nonmonotonic logic* (NML), Shoham [14] shows that the above theorem does not hold in general. He argues that this is due to the difference of the definitions of entailment between classical logic and nonmonotonic logic. In classical logic, a formula F is entailed by a theory T (i.e., $T \models F$) if F is satisfied in every model of T. In nonmonotonic logic, on the other hand, the corresponding relation $T \models_{NML} F$ is defined if F is satisfied in every *preferred* model of T. The set of preferred models of T is a subset of the set of all models of T, then a formula F which cannot be entailed in classical logic might be entailed in nonmonotonic logic.[3]

In this paper, we regard stable models as preferred models of a program. Then, the next relation holds.

Proposition 3.1 *Let P be a normal logic program. For any rule R, $P \models R$ implies $P \models_s R$, but not vice-versa.*

Proof. $P \models R$ means that R is satisfied in every model of P. Since stable models are models of P, $P \models R$ implies $P \models_s R$. It is enough to show a counter-example for the converse. Let $P = \{ a \leftarrow not\, b \}$ which has the unique stable model $\{a\}$. Then, $P \models_s a$ but $P \not\models a$. This is because P has the (non-stable) model $\{b\}$ in which a is false. □

Note that the converse does not hold in general even if a program is Horn.

Example 3.1. Let P be the program

$$a \leftarrow b$$

which has the stable model (or the least model) $\emptyset$. Then, $P \models_s c \leftarrow d$ but $P \not\models c \leftarrow d$. That is, $c \leftarrow d$ is satisfied in the least model of P, but it is not necessarily satisfied in every model of P. In fact, P has the model $\{d\}$ which does not satisfy $c \leftarrow d$.

Shoham presents that in nonmonotonic logic the deduction theorem is false in general, while the only-if direction still holds.[4] We first provide the corresponding result in normal logic programs. A *nested rule* is defined as

$$A \leftarrow R$$

where A is an atom and R is a rule of the form (4). An interpretation I satisfies a ground nested rule $A \leftarrow R$ if $I \models R$ implies $A \in I$. For a program P, $P \models_s (A \leftarrow R)$ if $A \leftarrow R$ is satisfied in every stable model of P.

[3] This characterizes the unique feature of NML that "jumping to a conclusion".
[4] Shoham provides no proof of this fact.

Theorem 3.2. *Let P be a normal logic program and R a rule such that $P \cup \{R\}$ is consistent. If $P \cup \{R\} \models_s A$, then $P \models_s A \leftarrow R$ for any ground atom A.*

Proof. When $P \cup \{R\} \models_s A$, A is true in every stable model of $P \cup \{R\}$. To see $P \models_s A \leftarrow R$, we show that for any stable model M of P, $M \models R$ implies $A \in M$. Suppose that there is a stable model M of P such that $M \models R$. Let R_g be any ground instance of R. Clearly, $M \models R$ implies $M \models R_g$, thereby $M \models {R_g}^M$. Since M is the least model of P^M and $M \models {R_g}^M$, M is a model of $(P^M \cup \{{R_g}^M\}) = (P \cup \{R\})^M$. Suppose that N is the least model of $(P \cup \{R\})^M$ s.t. $N \subseteq M$. As $(P \cup \{R\})^M$ is Horn and consistent, the least model of P^M becomes a subset of the least model of $(P \cup \{R\})^M$, i.e., $M \subseteq N$. Hence, $M = N$. Since M is the least model of $(P \cup \{R\})^M$, M is a stable model of $P \cup \{R\}$. As A is true in every stable model of $P \cup \{R\}$, $A \in M$ holds. Hence, $M \models R$ implies $A \in M$, and the result holds. □

The converse does not hold in general.

Example 3.2. Let P be the program

$$a \leftarrow not\ b$$

which has the unique stable model $\{a\}$. Then, $P \models_s a \leftarrow b$. On the other hand, $P \cup \{b\}$ has the unique stable model $\{b\}$, so $P \cup \{b\} \not\models_s a$.

An additional condition is necessary for the converse.

Theorem 3.3. *Let P be a normal logic program and R a rule such that $P \cup \{R\}$ is consistent. Then, if $P \models_s A \leftarrow R$ and $P \models_s R$, then $P \cup \{R\} \models_s A$ for any ground atom A.*

Proof. By $P \models_s R$, $M \models R$ for any stable model M of P. In this case, $P \models_s A \leftarrow R$ implies $A \in M$. Then, for any ground instance R_g of R, $M \models {R_g}^M$ holds. Since M is the least model of P^M and $M \models {R_g}^M$, M is a model of $(P^M \cup \{{R_g}^M\}) = (P \cup \{R\})^M$. Suppose that N is the least model of $(P \cup \{R\})^M$ s.t. $N \subseteq M$. Then, we can show $M = N$ in the same manner as Theorem 3.2. Thus, $A \in M$ for the least model M of $(P \cup \{R\})^M$. This implies that $A \in M$ for any stable model M of $P \cup \{R\}$. Hence, $P \cup \{R\} \models_s A$. □

We call Theorems 3.2 and 3.3 the *entailment theorem* of normal logic programs.[5]

4 Contrapositive Rules

In normal logic programs, a rule in a program is not a "clause" in the classical sense because of the presence of negation as failure. This makes the meaning of

[5] We avoid using the term "deduction" because the entailment symbol $\models_s$ does not imply deductive consequences in the classical sense.

rules different from the one in classical logic. One of such distinctive features is that a rule and its *contrapositive* with respect to the implication $\leftarrow$ and negation *not* are not semantically equivalent. For example, the program

$$a \leftarrow$$

has the stable model $\{a\}$, while the program which consists of its contrapositive

$$\leftarrow \mathit{not}\, a$$

has no stable model.

However, in the process of inverse entailment a rule must be contraposed, hence the introduction of contrapositive rules is necessary to apply inverse entailment to normal logic programs. This section introduces a theory of contrapositive rules in normal logic programs and presents their properties.

Definition 4.1. Given a rule R of the form:

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\, A_{m+1}, \ldots, \mathit{not}\, A_n\,,$$

its *contrapositive rule* ${}^cR$ is defined as

$$\mathit{not}\, A_1\,; \cdots ;\, \mathit{not}\, A_m;\, \mathit{not}\, \mathit{not}\, A_{m+1}\,; \cdots ;\, \mathit{not}\, \mathit{not}\, A_n \;\leftarrow\; \mathit{not}\, A_0 \tag{6}$$

where ";" means disjunction. The left-hand side of $\leftarrow$ is the *head* and the right-hand side is the *body* of the rule.

In the contrapositive rule, the atoms A_0, $A_1, \ldots, A_m$ in the original rule are shifted to the other side of the implication with the introduction of *not*. On the other hand, the NAF formulas $\mathit{not}\, A_{m+1}, \ldots, \mathit{not}\, A_n$ in the body are shifted to the head of the rule with the introduction of "nested NAF". The semantics of such nested NAF is given in [8]. A ground rule of the form (6) is *satisfied* in an interpretation I if $A_0 \notin I$ implies either $\{A_1, \ldots, A_m\} \setminus I \neq \emptyset$ or $\{A_{m+1}, \ldots, A_n\} \cap I \neq \emptyset$ (written as $I \models {}^cR$). When a rule ${}^cR$ contains variables, $I \models {}^cR$ means that I satisfies every ground instance of ${}^cR$.

According to [8], the rule (6) is equivalent to the integrity constraint

$$\leftarrow A_1, \ldots, A_m, \mathit{not}\, A_{m+1}, \ldots, \mathit{not}\, A_n, \mathit{not}\, A_0 \tag{7}$$

under the stable model semantics. Hence we identify the contrapositive rule (6) with the integrity constraint (7) hereafter. In particular,

$$\mathit{not}\, A \leftarrow$$

is identified with

$$\leftarrow A,$$

and

$$\mathit{not}\, \mathit{not}\, A \leftarrow$$

is identified with

$$\leftarrow not\ A.$$

The contrapositive form of a nested rule is defined as follows. A *nested rule* R considered here is of the form:

$$L \leftarrow Q$$

where L is an LP-literal and Q is a rule of the form (4). An interpretation I satisfies a ground nested rule $L \leftarrow Q$ if $I \models Q$ implies $I \models L$. For a program P, $P \models_s (L \leftarrow Q)$ if $L \leftarrow Q$ is satisfied in every stable model of P. Then, a *contrapositive rule* ${}^cR$ is defined as

$$not\, Q \leftarrow not\, L\,. \tag{8}$$

A ground rule of the form (8) is *satisfied* in an interpretation I if $I \not\models L$ implies $I \not\models Q$. In particular, a ground rule of the form

$$not\, Q \leftarrow$$

is satisfied in I if $I \not\models Q$.

For a program P, $P \models_s {}^cR$ if ${}^cR$ is satisfied in every stable model of P.

Theorem 4.1. *Let P be a normal logic program, R a (nested) rule, and ${}^cR$ its contrapositive. Then, $P \models_s R$ iff $P \models_s {}^cR$.*

Proof. $P \models_s R$ iff $P \models_s R_g$ for any ground instance R_g of R.
First, let R_g be a ground (unnested) rule of the form
$A_0 \leftarrow A_1, \ldots, A_m, not\, A_{m+1}, \ldots, not\, A_n$. Then, $P \models_s R_g$
iff for any stable model M of P, $\{A_1, \ldots, A_m\} \subseteq M$ and $\{A_{m+1}, \ldots, A_n\} \cap M = \emptyset$ imply $A_0 \in M$
iff for any stable model M of P, $A_0 \notin M$ implies either $\{A_1, \ldots, A_m\} \setminus M \neq \emptyset$ or $\{A_{m+1}, \ldots, A_n\} \cap M \neq \emptyset$
iff $P \models_s {}^cR_g$
iff $P \models_s {}^cR$.

Next, let R_g be a ground nested rule of the form $L \leftarrow Q$. Then, $P \models_s R_g$
iff for any stable model M of P, $M \models Q$ implies $M \models L$
iff for any stable model M of P, $M \not\models L$ implies $M \not\models Q$
iff $P \models_s {}^cR_g$
iff $P \models_s {}^cR$. □

5 Inverse Entailment in Normal Logic Programs

We first apply the entailment theorem (Theorems 3.2 and 3.3) to a ground LP-literal *not A*.

Theorem 5.1. *Let P be a normal logic program and R a rule such that $P \cup \{R\}$ is consistent. If $P \cup \{R\} \models_s not\ A$, then $P \models_s (not\ A \leftarrow R)$ for any ground atom A. In converse, if $P \models_s (not\ A \leftarrow R)$ and $P \models_s R$, then $P \cup \{R\} \models_s not\ A$.*

Proof. If $P \cup \{R\} \models_s not\ A$, A is false in every stable model of $P \cup \{R\}$. Then, the rest of the proof in this direction proceeds in the same manner as Theorem 3.2. On the other hand, if $P \models_s (not\ A \leftarrow R)$ and $P \not\models_s R$, $A \notin M$ for any stable model M of P. Then, for any ground instance R_g of R, $M \models {R_g}^M$ and $A \notin M$ for the least model M of P^M. Then, we can show that M is the least model of $(P \cup \{R\})^M$ as in Theorem 3.3. This implies that $A \notin M$ for any stable model M of $P \cup \{R\}$. Hence, $P \cup \{R\} \models_s not\ A$. □

Proposition 5.2 *Let P be a normal logic program and L a ground LP-literal. If $P \models_s not\ L$, then $P \cup \{\, not\ L \,\}$ is consistent.*[6]

Proof. $P \models_s not\ L$ implies $P \models_s \leftarrow L$. Then, $M \models \leftarrow L$ holds for every stable model M of P. Let M be any stable model of P. Then, M is the least model of P^M and $M \models \leftarrow L$ iff M is the least model of $(P \cup \{\, \leftarrow L \,\})^M$ iff M is a stable model of $P \cup \{\, \leftarrow L \,\}$. Hence, $P \cup \{\, not\ L \,\}$ is consistent. □

Now we are ready to introduce inverse entailment in normal logic programs.

Theorem 5.3. *Let P be a normal logic program and R a rule such that $P \cup \{R\}$ is consistent. For any ground LP-literal L, if $P \cup \{R\} \models_s L$ and $P \models_s not\ L$, then $P \cup \{\, not\ L \,\} \models_s not\ R$ where $P \cup \{\, not\ L \,\}$ is consistent.*

Proof. If $P \cup \{R\} \models_s L$, then $P \models_s (L \leftarrow R)$ (Theorems 3.2 and 5.1). On the other hand, $P \models_s (L \leftarrow R)$ iff $P \models_s (not\ R \leftarrow not\ L)$ (Theorem 4.1). Put $A = R$, where A is an atom $p(x)$ such that p appears nowhere in P and x is a vector of variables appearing in R. Then, $P \models_s (not\ R \leftarrow not\ L)$ iff $P \models_s (not\ A \leftarrow not\ L)$. By $P \models_s (not\ A \leftarrow not\ L)$ and $P \models_s not\ L$, it holds that $P \cup \{\, not\ L \,\} \models_s not\ A$ (Theorems 5.1). Thus, $P \cup \{\, not\ L \,\} \models_s not\ R$. Also, $P \models_s not\ L$ implies the consistency of $P \cup \{\, not\ L \,\}$ (Proposition 5.2). Hence, the result holds. □

The converse of Theorem 5.3 does not hold in general.

Example 5.1. Let P be the program

$$a \leftarrow b$$

with $L = c$, and $R = (b \leftarrow)$. In this case, $P \cup \{\, not\ L \,\}$ becomes

$$a \leftarrow b,$$
$$\leftarrow c,$$

where $not\ c$ is expressed as the constraint $\leftarrow c$. This program has the stable model $\emptyset$ in which R is false, so that $P \cup \{\, not\ L \,\} \models_s not\ R$. On the other hand, $P \cup \{R\}$ becomes

$$a \leftarrow b,$$
$$b \leftarrow,$$

which has the stable model $\{\, a, b \,\}$. Since $L = c$ is false in this stable model, $P \cup \{R\} \not\models_s L$.

[6] $\{\, not\ L \,\}$ is an abbreviation of $\{\, not\ L \leftarrow \,\}$ which is equivalent to $\{\, \leftarrow L \,\}$.

Thus, the relation $P \cup \{ not\ L \} \models_s not\ R$ provides a necessary (but not always sufficient) condition for computing a rule R satisfying $P \cup \{R\} \models_s L$ and $P \models_s not\ L$.

Next we present a method of applying the above new inverse entailment theorem to computing inductive hypotheses in normal logic programs. We first characterize an induction problem considered here.

Given :
- a *background knowledge base* P as a *categorical* normal logic program,
- a ground LP-literal L such that $P \models_s not\ L$, where L represents a *positive example* in case of $L = A$ with a ground atom A; otherwise it represents a *negative example* in case of $L = not\ A$,
- a *target predicate* which is subject to learn.

Find : a hypothetical rule R such that
- $P \cup \{R\} \models_s L$,
- $P \cup \{R\}$ is consistent,
- R has the target predicate in its head.

Here, we assume that P, R, and L have the same language. In the above, the assumption $P \models_s not\ L$ presents that the given example is false in the unique stable model of the background program.[7] In fact, if the example L is true in P, there is no need to find R.

First, suppose that there is a rule R such that $P \cup \{R\}$ is consistent and $P \cup \{R\} \models_s L$. When $P \models_s not\ L$, it holds by Theorem 5.3 that

$$P \cup \{ not\ L \} \models_s not\ R \tag{9}$$

where $P \cup \{ not\ L \}$ is consistent.

The relation (9) means that R is not satisfied in the stable model of $P \cup \{ not\ L \}$. Here we assume that P has the unique stable model and $P \models_s not\ L$. Then, the next proposition holds.

Proposition 5.4 *Let P be a categorical normal logic program. Also, let L be a ground LP-literal such that $P \models_s not\ L$. Then, the stable model of P is equivalent to the stable model of $P \cup \{ not\ L \}$.*

Proof. M is the stable model of P s.t. $P \models_s not\ L$
iff M is the least model of P^M and $M \models \leftarrow L$
iff M is the least model of $(P \cup \{ \leftarrow L \})^M$
iff M is the stable model of $P \cup \{ \leftarrow L \}$
iff M is the stable model of $P \cup \{ not\ L \}$. □

Using Proposition 5.4, the relation (9) is simplified as

$$P \models_s not\ R\,. \tag{10}$$

[7] If $L = not\ A$, $P \models_s not\ not\ A$ means that A is true in the stable model of P.

Next, put

$$M^+ = \{\, l \mid l \in \mathcal{HB}^+ \text{ and } P \models_s l \,\}.$$

M^+ is the set of LP-literals, which augments the stable M of P by the LP-literals $\{\, not\, A \mid A \in \mathcal{HB} \setminus M \,\}$. We call M^+ the *expansion* of M. Then, by (10) it holds that

$$M^+ \models not\, R\,. \tag{11}$$

Let r_0 be an integrity constraint $\leftarrow \Gamma$ where Γ is a conjunction of ground LP-literals such that $\Gamma \subseteq M^+$. Since M^+ does not satisfy r_0, r_0 satisfies the relation (11), i.e., $M^+ \models not\, r_0$.

Here we define a couple of notions. Given two ground LP-literals L_1 and L_2, we define the relation $L_1 \sim L_2$ if L_1 and L_2 have the same predicate and $const(L_1) = const(L_2)$ where $const(L)$ denotes the set of constants appearing in L. Let L_1 and L_2 be two ground LP-literals such that each literal has a predicate with more than 0 argument. Then, L_1 in a ground rule R is *relevant* to L_2 if either (i) $L_1 \sim L_2$ or (ii) L_1 shares a constant with an LP-literal L_3 in R such that L_3 is relevant to L_2. Otherwise, L_1 is *irrelevant* to L_2.

Now we apply the following series of transformations to the above r_0.

1. (dropping irrelevant atoms): Drop any LP-literal from r_0 which is irrelevant to the given example. The resulting rule is denoted as r_1.
2. (dropping implied atoms): Suppose that the body of r_1 contains both an atom A and the conjunction B of LP-literals such that $A \neq B$, and the rule $A \leftarrow B$ is in the ground instantiation of the background program. Then, drop A from the body of r_1. The resulting rule is denoted as r_2.
3. (shifting the target atom): Let $r_2 = \leftarrow \Gamma$ where Γ is a conjunction of ground LP-literals. If Γ contains an LP-literal $not\, A$ with the target predicate, produce the rule $r_3 : A \leftarrow \Gamma'$ where $\Gamma' = \Gamma \setminus \{not\, A\}$.
4. (generalization): Replace constants appearing in r_3 by appropriate variables. That is, construct a rule r_4 such that $r_4\theta = r_3$ for some substitution θ.

The above transformation sequence 1–4 is denoted as $Trans$. The next theorem presents that any rule produced by $Trans$ satisfies the relation (11).

Theorem 5.5. *Let M^+ be a set of ground LP-literals defined as above. If R_{ie} is a rule which is obtained by $Trans$, then $M^+ \models not\, R_{ie}$ holds.*

Proof. Let $M^+ \models not\, r_0$. Dropping any ground LP-literals from the body of the constraint r_0 does not violate this relation. Hence, $M^+ \models not\, r_1$ and $M^+ \models not\, r_2$. Next, let $r_3 : A \leftarrow \Gamma'$ which is obtained from r_2 by converting $not\, A$ in the body of r_2 to A in the head of r_3. By $M^+ \models not\, r_2$, $\Gamma' \subseteq M^+$ and $not\, A \in M^+$. Hence, $M^+ \models not\, r_3$. Finally, consider r_4. Since M^+ does not satisfy a ground instance r_3 of r_4, $M^+ \not\models r_4$ thereby $M^+ \models not\, r_4$. Hence, the result holds. □

When $P \cup \{R_{ie}\}$ is consistent, we say that a rule R_{ie} is computed using the *inverse entailment rule* (*IE rule*, for short) from an LP-literal L in the program P (under the stable model semantics).

6 Examples

Example 6.1. Let P be the background program:

$$\begin{aligned} &bird(x) \leftarrow penguin(x), \\ &bird(tweety) \leftarrow, \\ &penguin(polly) \leftarrow . \end{aligned}$$

Given the positive example $L = flies(tweety)$, $P \models_s not\, flies(tweety)$. Then, the set M^+ of LP-literals becomes

$$\begin{aligned} M^+ = \{ &bird(tweety),\, bird(polly),\, penguin(polly), \\ &not\, penguin(tweety),\, not\, flies(tweety),\, not\, flies(polly)\,\}. \end{aligned}$$

From M^+ the integrity constraint:

$$\begin{aligned} r_0 : \leftarrow\ &bird(tweety),\, bird(polly),\, penguin(polly), \\ &not\, penguin(tweety),\, not\, flies(tweety),\, not\, flies(polly) \end{aligned}$$

is constructed.

Next we apply $Trans$. First, LP-literals which are irrelevant to the example L are dropped:

$$r_1 : \leftarrow bird(tweety),\, not\, penguin(tweety),\, not\, flies(tweety).$$

When $flies$ is the target predicate, shifting $flies(tweety)$ to the head produces

$$r_3 :\ flies(tweety) \leftarrow bird(tweety),\, not\, penguin(tweety)\,.$$

Finally, replacing $tweety$ by a variable x, we get

$$R_{ie} :\ flies(x) \leftarrow bird(x),\, not\, penguin(x)$$

where $P \cup \{R_{ie}\} \models_s L$ holds.

Example 6.2. Let P be the background program:

$$\begin{aligned} &flies(x) \leftarrow bird(x),\, not\, ab(x), \\ &bird(x) \leftarrow penguin(x), \\ &bird(tweety) \leftarrow, \\ &penguin(polly) \leftarrow . \end{aligned}$$

Given the negative example $L = not\, flies(polly)$, $P \models_s not\, not\, flies(polly)$ holds. Then, the set M^+ of LP-literals becomes

$$\begin{aligned} M^+ = \{ &bird(tweety),\, bird(polly),\, penguin(polly), \\ &not\, penguin(tweety),\, flies(tweety),\, flies(polly), \\ &not\, ab(tweety),\, not\, ab(polly)\,\}. \end{aligned}$$

Suppose that the following integrity constraint is constructed from M^+.

$$r_0 : \leftarrow bird(tweety),\ bird(polly),\ penguin(polly),\\ not\ penguin(tweety),\ flies(tweety),\ flies(polly),\\ not\ ab(tweety),\ not\ ab(polly)\,.$$

By dropping LP-literals which are irrelevant to L, it becomes

$$r_1 : \leftarrow bird(polly),\ penguin(polly),\ flies(polly), not\ ab(polly)\,.$$

As $bird(polly)$ is implied by $penguin(polly)$ in P, it is dropped as

$$r_2 : \leftarrow penguin(polly),\ flies(polly), not\ ab(polly)\,.$$

Now let ab be the target predicate. Then, shifting $ab(polly)$ to the head, it becomes

$$r_3 :\ ab(polly) \leftarrow penguin(polly),\ flies(polly)\,.$$

Replacing $polly$ by a variable x, we get

$$r_4 :\ ab(x) \leftarrow penguin(x),\ flies(x)\,.$$

However, $P \cup \{r_4\}$ is inconsistent (i.e., having no stable model).

To get a consistent program, construct r_0' as

$$r_0' : \leftarrow bird(tweety),\ bird(polly),\ penguin(polly),\\ not\ penguin(tweety),\ not\ ab(tweety),\ not\ ab(polly)\,.$$

Applying $Trans$ to this r_0', we get

$$R_{ie} :\ ab(x) \leftarrow penguin(x)$$

where $P \cup \{R_{ie}\} \models_s L$ holds.

Our IE rule is different from Muggleton's one even in the Horn cases.

Example 6.3. [15] Let P be the background program:

$$even(s(x)) \leftarrow odd(x),\\ even(0) \leftarrow,$$

and the positive example $L = odd(s^3(0))$ where $P \models_s not\ L$. Then, it becomes

$$M^+ = \{\, even(0),\ not\ even(s(0)),\ \ldots\ ,not\ odd(0),\ not\ odd(s(0)),\ \ldots\,\}.$$

Take the constraint r_0 as

$$\leftarrow even(0),\ not\ odd(s(0)).$$

By shifting $odd(s(0))$ to the head and generalizing it, the rule

$$R_{ie} :\ odd(s(x)) \leftarrow even(x)$$

is obtained. Recall that Muggleton's IE rule cannot derive this rule.[8]

[8] In a Horn logic program, M^+ has the effect which is similar to Muggleton's *enlarged bottom set* [12].

7 Discussion

The present ILP systems mostly consider Horn logic programs as background theories, and inverse entailment is used for finding hypotheses efficiently in a program. On the other hand, Horn logic programs have limitation in representing incomplete knowledge and reasoning with commonsense. This is because commonsense reasoning with incomplete knowledge is inherently nonmonotonic. This observation led to the extensive study of nonmonotonic extensions of logic programming, which includes logic programs with negation as failure [2].

Extending the representation language and enhancing reasoning ability are also indispensable for constructing a powerful ILP system. In this sense, combining induction and commonsense reasoning in the framework of *nonmonotonic ILP* is an important step in the ILP research. However, the present ILP techniques in Horn logic programs are not always applicable in their present form. For example, it is shown in [13] that inverse resolution causes some problems in the presence of NAF. Inverse entailment is also an important technique in Horn ILP, so it is meaningful to examine the technique in the nonmonotonic situation and to reconstruct a theory. There are some ILP systems which handle nonmonotonic logic programs as background theories (e.g. [1,4,3,10,7,5]). To our best knowledge, however, no study tackles the problem of reformulating IE in nonmonotonic theories.

An inductive inference rule is *correct* if it produces a rule R satisfying $P \cup \{R\} \models_s L$ for a program P and an example L. In contrast to the IE rule in Horn logic programs, the correctness of our IE rule is susceptible to the construction of R. For one reason, in nonmonotonic logic programs a rule is viewed as a derivation rule and its representation form is meaningful. For example, given the program $P = \{\, a \leftarrow,\ \ b \leftarrow b \,\}$ and the positive example $L = c$, it becomes $M^+ = \{\, a, \mathit{not}\, b, \mathit{not}\, c \,\}$. Then, from the integrity constraint $\leftarrow a, \mathit{not}\, b, \mathit{not}\, c$, two different rules $R_1 = (c \leftarrow a, \mathit{not}\, b)$ and $R_2 = (b \leftarrow a, \mathit{not}\, c)$ are constructed according to the different choice of the target predicate. In this case, both R_1 and R_2 satisfy the relation $M^+ \models \mathit{not}\, R$, while $P \cup \{R_1\} \models_s L$ but $P \cup \{R_2\} \not\models_s L$. The failure of R_2 is due to the inappropriate selection of the target predicate. If we choose c as the target, the IE rule produces the correct rule R_1. In many cases the target predicate is the predicate in the given example, while this is not always the case as in Example 6.2.

On the other hand, the transformation sequence $Trans$ outputs rules satisfying $M^+ \models \mathit{not}\, R$, while $Trans$ also filters out useless hypotheses. For instance, in Example 6.1 the rule $\mathit{flies}(\mathit{tweety}) \leftarrow \mathit{bird}(\mathit{polly})$ explains $\mathit{flies}(\mathit{tweety})$ in P, but this rule is dropped in the process of $Trans$. In this sense, our IE rule is not necessarily *complete* for the computation of rules satisfying $P \cup \{R\} \models_s L$. However, we are dubious about the practical value of the completeness of an inductive inference rule. This is because there may exist possibly infinite solutions for explaining given examples in general, and it seems meaningless to guarantee the completeness for computing tons of useless hypotheses. What is important is selecting meaningful hypotheses in the process of computation, and $Trans$ realizes this kind of filtering. We showed in Example 6.3 that a meaningful hy-

pothesis, which is not computed by Muggleton's IE rule, is obtained using our IE rule.

Finally, we remark the computational complexity of our IE rule in the propositional case. Given a finite propositional normal logic program P, checking whether an interpretation M of P is a stable model is done in polynomial time.[9] Thus, checking whether a set M^+ of LP-literals is the expansion of the stable model of a categorical program is also executed in polynomial time. Our IE rule selects a subset of M^+ and composes R_{ie}. This task requires exponential computation in general, however, the transformation $Trans$ reduces the number of candidate hypotheses, which would ease the construction of R_{ie}.

8 Summary

This paper introduced an inverse entailment rule in nonmonotonic ILP. We provided a new entailment theorem and a contrapositive theorem to construct a theory of inverse entailment. We demonstrated that the new inverse entailment rule successfully finds appropriate inductive hypotheses in normal logic programs. The proposed method extends the present technique to a syntactically and semantically richer framework, and contributes to a theory of nonmonotonic ILP.

In this paper, we considered the IE rule for categorical programs having exactly one stable model. When a program has multiple stable models, however, the proposed technique is inapplicable in its present form. It is necessary to relax the condition for further extension.

Acknowledgements

The author thanks Katsumi Inoue for recalling Shoham's result. He also thanks Akihiro Yamamoto for clarifying the problem of Muggleton's IE rule.

References

1. M. Bain and S. Muggleton. Non-monotonic learning. In: S. Muggleton (ed.) *Inductive Logic Programming*, Academic Press, pp. 145–161, 1992.
2. C. Baral and M. Gelfond. Logic programming and knowledge representation. *Journal of Logic Programming* 19/20:73–148, 1994.
3. F. Bergadano, D. Gunetti, M. Nicosia, and G. Ruffo. Learning logic programs with negation as failure. In: L. De Raedt (ed.) *Advances in Inductive Logic Programming*, IOS Press, pp. 107–123, 1996.
4. Y. Dimopoulos and A. Kakas. Learning nonmonotonic logic programs: learning exceptions. In: *Proceedings of the 8th European Conference on Machine Learning, Lecture Notes in Artificial Intelligence* 912, Springer-Verlag, pp. 122–137, 1995.

[9] More precisely, it takes polynomial time to derive P^M and to compute the least model of P^M. Checking the equivalence between M and the least model is also done in polynomial time.

5. L. Fogel and G. Zaverucha. Normal programs and multiple predicate learning. In: *Proceedings of the 8th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1446, Springer-Verlag, pp. 175–184, 1998.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In: *Proceedings of the 5th International Conference and Symposium on Logic Programming*, MIT Press, pp. 1070–1080, 1988.
7. K. Inoue and Y. Kudoh. Learning extended logic programs. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 176–181, 1997.
8. V. Lifschitz, L. R. Tang, and H. Turner. Nested expression in logic programs. *Annals of Mathematics and Artificial Intelligence* 25:369–389, 1999.
9. J. W. Lloyd. *Foundations of logic programming* (2nd edition). Springer-Verlag, 1987.
10. L. Martin and C. Vrain. A three-valued framework for the induction of general logic programs. In: L. De Raedt (ed.) *Advances in Inductive Logic Programming*, IOS Press, pp. 219–235, 1996.
11. S. Muggleton. Inverse entailment and Progol. *New Generation Computing* 13:245–286, 1995.
12. S. Muggleton. Completing inverse entailment. In: *Proceedings of the 8th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1446, Springer-Verlag, pp. 245–249, 1998.
13. C. Sakama. Some properties of inverse resolution in normal logic programs. In: *Proceedings of the 9th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1634, Springer-Verlag, pp. 279–290, 1999.
14. Y. Shoham. Nonmonotonic logics: meaning and utility. In: *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, pp. 388–393, 1987.
15. A. Yamamoto. Which hypotheses can be found with inverse entailment? In: *Proceedings of the 7th International Workshop on Inductive Logic Programming, Lecture Notes in Artificial Intelligence* 1297, Springer-Verlag, pp. 296–308, 1997.

A Note on Two Simple Transformations for Improving the Efficiency of an ILP System

Vítor Santos Costa[1], Ashwin Srinivasan[2], and Rui Camacho[3]

[1] COPPE/Sistemas, UFRJ, Brazil and LIACC, Universidade do Porto, Portugal
[2] Oxford University Comp. Lab., Wolfson Bldg., Parks Rd, Oxford, UK
[3] LIACC and FEUP, Universidade do Porto, Portugal

Abstract. Inductive Logic Programming (ILP) systems have had noteworthy successes in extracting comprehensible and accurate models for data drawn from a number of scientific and engineering domains. These results suggest that ILP methods could enhance the model-construction capabilities of software tools being developed for the emerging discipline of "knowledge discovery from databases." One significant concern in the use of ILP for this purpose is that of efficiency. The performance of modern ILP systems is principally affected by two issues: (1) they often have to search through very large numbers of possible rules (usually in the form of definite clauses); (2) they have to score each rule on the data (usually in the form of ground facts) to estimate "goodness". Stochastic and greedy approaches have been proposed to alleviate the complexity arising from each of these issues. While these techniques can result in order-of-magnitude improvements in the worst-case search complexity of an ILP system, they do so at the expense of exactness. As this may be unacceptable in some situations, we examine two methods that result in admissible transformations of clauses examined in a search. While the methods do not alter the size of the search space (that is, the number of clauses examined), they can alleviate the theorem-proving effort required to estimate goodness. The first transformation simply involves eliminating literals using a weak test for redundancy. The second involves partitioning the set of literals within a clause into groups that can be executed independently of each other. The efficacy of these transformations are evaluated empirically on a number of well-known ILP datasets. The results suggest that for problems that require the use of highly non-determinate predicates, the transformations can provide significant gains as the complexity of clauses sought increases.

1 Introduction

An early Knuth classic [12] describes how the use of run-time statistics gathered from the execution of a program can be used to suggest optimisations. The paper is an empirical study of executing FORTRAN programs selected randomly from semi-protected files stored on Stanford University's computer disks[1]. He summarises the results of the study thus:-

[1] Knuth describes two other less successful methods to obtain data for his experiments. These were: rummaging waste-baskets and recycling bins; and posting a sentry at

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 225–242, 2000.

> In the early days of computing, machine time was king, and people worked hard to get extremely efficient programs. Eventually machines got larger and faster, and the pay off for writing fast programs was measured in minutes or seconds instead of hours... Therefore most of the emphasis in software development has been in making programs easier to write, easier to understand and easier to change. There is no doubt that this emphasis has reduced the total system costs in many installations, but there is also little doubt that the corresponding lack of emphasis on efficient code has resulted in systems which can be greatly improved... Frequency counts give an important dimension to programs, showing programmers how to make their routines more efficient with comparatively little effort.

By "frequency counts" Knuth is referring here to the frequency with which some recognisable constructions actually occur during execution of a FORTRAN program. He outlines two desirable outcomes of such studies. First, that dynamic profiles of program execution can direct the programmer to simple optimisations that can result in more efficient code. Second, that such profiles (provided they are representative of typical profiles, of course) can direct the design of compilers by indicating what programmers want to do. In this paper we illustrate, on a smaller scale, an instance of realising the first of these outcomes within the province of Inductive Logic Programming (ILP).

Since 1991, ILP systems have been used to construct predictive models for data drawn from diverse domains. These include software analysis [2], engineering [7, 9], environment monitoring [8], biochemistry [10, 11, 16], and language processing [4, 27]. The systems that have been used to construct these models have—at a very coarse level—been fairly similar, namely: search engines that repeatedly examine sets of candidates (the "search space") to find good rules. The time-complexity of most of these systems therefore largely depends on: (1) the size of the search space; and (2) the cost of evaluating the "goodness" of a rule. Recently, stochastic methods have been proposed for reducing the complexity of both these aspects (for example, see [20, 21, 22]), at the expense of exactness. That is, lower complexity ILP systems are possible if we are willing to accept sub-optimal rules, or imprecise estimates of the goodness of a rule. Such compromises may not always be acceptable, and we present here two simple transformations that are suggested by examining the dynamic profile obtained by using an ILP system in a fairly conventional manner to construct a model for data. Both transformations do not alter the number of rules examined; but are shown empirically, in some cases, to alter significantly the cost of evaluating a rule.

The paper is organised as follows. In Section 2 we outline a simple procedure constituting an ILP algorithm. This constructs models for data using the greedy set-covering approach common to many extant implementations. Section 3 tabulates the dynamic profile of using an ILP system implementing this procedure,

a card-reader to accost programmers. The former was found to largely yield non-working programs, and the latter too time-consuming to be practical.

when constructing a model on a benchmark dataset. Section 4 describes the two transformations. Section 5 contains an empirical study of the efficacy of the transformations, using some well-known datasets. Section 6 concludes the paper. Prolog code implementing the transformations is in Appendix A.

2 A Simple ILP Implementation

We adopt the following informal prescription for an ILP algorithm (based on [14]). Given: (a) background knowledge B in the form of a Prolog program; (b) some language specification $\mathcal{L}$ describing the hypotheses; (c) an optional set of constraints I on acceptable hypotheses; and (d) a finite set of examples E at least some of whose elements constitute a set of "positive" examples E^+ such that none of the E^+ are derivable from B; the task is to find some hypothesis H in $\mathcal{L}$, such that: (1) H is consistent with the I; and (2) The E^+ are derivable from B, H.

Several prominent ILP systems (for example, CProgol [15]; Golem [17]; and FOIL [19]) use a simple iterative procedure to construct such a hypothesis one clause at a time. Figure 1 shows one version of such a procedure. This is reproduced with minor changes from [22] and we refer to the reader to that paper for proofs of correctness and complexity arguments. Here we will assume that: (a) on each iteration, $search(\cdots)$ will examine a finite number of clauses; (b) the best clause amongst these alternatives requires $search(\cdots)$ to at least check the derivability of each example in $Train_i$; and (c) the check for derivability terminates.

$generalise(B, I, \mathcal{L}, E)$: Given background knowledge B, hypothesis constraints I, a finite training set $E = E^+ \cup E^-$, returns a hypothesis H in $\mathcal{L}$ such that $B \cup H$ derives the E^+.

1. $i = 0$
2. $E_i^+ = E^+$, $H_i = \emptyset$
3. while $E_i^+ \neq \emptyset$ do
   (a) increment i
   (b) $Train_i = E_{i-1}^+ \cup E^-$
   (c) $D_i = search(B, H_{i-1}, I, \mathcal{L}, Train_i)$
   (d) $H_i = H_{i-1} \cup \{D_i\}$
   (e) $E_p = \{e_p : e_p \in E_{i-1}^+ \ s.t. \ B \cup H_i \models \{e_p\}\}$.
   (f) $E_i^+ = E_{i-1}^+ \setminus E_p$
4. return H_i

Fig. 1. A simple ILP implementation. The function $search(\cdots)$ is some search procedure that returns one clause after evaluating alternatives in some search-space of clauses, all of which are in $\mathcal{L}$.

3 Dynamic Statistics

Knuth describes two different types of statistics obtainable from the execution of programs by a compiler. These broadly relate to what the compiler is doing during program execution and why it is doing it (that is, which constructs in the programs are responsible for the main activities of the compiler). These statistics may then suggest optimisations that alleviate obvious execution bottlenecks. Here is one way to adopt this approach within ILP: sample from the space of ILP-system/ILP-datasets pairs. For each such pair, obtain dynamic statistics of the form just described. An average computed over all such profiles may then lead to a consideration of useful optimisations.

The principal practical difficulty in adopting the approach just outlined is the nature of current ILP implementations. These are encoded in a variety of different programming languages, each employing quite different constructs. As a first step therefore, we adopt the following more pragmatic approach: select a single ILP system (preferably one which is as general-purpose as possible) written in Prolog; execute the ILP system using a standard Prolog engine on one or more datasets; and examine dynamic profiles using the tools provided by the Prolog engine. There are two principal advantages in this. First, by selecting an ILP system written in Prolog, the same profiling tools can be applied to the execution of the hypothesis as well as the learning program itself[2]. Second, modern compilers or interpreters for the Prolog language appear to have converged to relatively uniform constructs and have good tools for constructing dynamic profiles.

Here we have selected the ILP system Aleph[3]. This is a Prolog program that can emulate a number of different ILP methods. We use Aleph to execute the simple clause-by-clause search mentioned earlier, using a search function that performs a restricted form of breadth-first branch-and-bound search (the restrictions ensure that the search only examines a finite number of alternatives). Further, every clause in the search is scored using an evaluation function that is simply the difference in the number of positive and negative examples derived by the clause (in conjunction with the background knowledge and clauses found earlier). For each clause in the search, this requires the search function to check derivability over all the examples. Aleph uses YAP[4], a high-performance Prolog compiler developed at The Universidade do Porto. To ensure derivability checks terminate, we use a special feature provided within YAP that allows depth-bounded evaluation. We shall use two tools that provide the kind of dynamic information described by Knuth. The first tool is the program `gprof`, an Unix tool that can be used to describes how the YAP Prolog compiler divides execution time. The second tool is a high-level profiler included in YAP. This profiler counts calls to procedures made by the user program (here, the ILP system Aleph).

[2] We are indebted to a referee of this paper for pointing this out.
[3] `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/`
[4] `http://www.ncc.up.pt/~vsc/Yap/`

We are still left with a choice of datasets. As a first step, we obtain an execution-profile from a single non-trivial application. We have elected to use the data concerning the prediction of carcinogenicity tests on rodents [23]. This dataset has a number of attractive features: it is an important practical problem; the background knowledge consists of large numbers of non-determinate predicate definitions (these can be a source of inefficiency when checking the derivability of an example); and our previous experience suggests that a fairly large search space needs to be examined to obtain a good clause. While obtaining dynamic statistics using a single ILP system and a single dataset is a significant compromise on the ideal presented at the outset of this section, the results are nevertheless interesting. Figure 2 summarises the information obtained from `gprof`.

Function	Involved in	Time (s)
`absmi`	WAM emulator	26110
`p_flags`	Meta-calls	840
`PredProp`	Meta-calls	830
`p_execute0`	Meta-calls	740
`a_eq`	Meta-calls	690
`a_le`	Meta-calls	450
`p_pred_goal`	Meta-calls	440
`FloatOfTerm`	Miscellaneous	1120
`CallProlog`	Miscellaneous	710
`GetExpProp`	Miscellaneous	630
`IUnify_complex`	Miscellaneous	460
`Eval`	Arithmetic expression evaluation	2040
`i_recorded`	Internal database access	440

Fig. 2. Prolog compiler activities when the ILP system Aleph analyses carcinogenesis outcome data. The profile is a summary of execution times (rounded up to the nearest 10s) reported by the `gprof` program accompanying the records the activities of the YAP compiler.

In Fig. 2 the proportion of times for the main activities are approximately: 74% for WAM (Warren Abstract Machine) emulation; 11% on "meta-calls" (explained shortly); 8% on miscellaneous activities; 6% on arithmetic evaluation; and 1% on internal database accesses. The high proportion of time allocated to WAM emulation simply means that most of the time is not spent in built-ins written in `C`, but in either use-code or built-ins written in Prolog. This is not surprising. What is unexpected is the high proportion of time spent on executing meta-calls. These are part of the compiler implementation of the Prolog built-in predicate `call/1`. Frequency counts obtained from the high-level YAP profiler

(not shown here due to space restrictions) clearly indicate that the principal source of meta-call activity is the check for example derivability. For each clause of the form $Head : -Body$, this involves: (a) matching $Head$ against an example (positive or negative); and (b) a meta-call to $Body$. Typically, $Body$ is a conjunction of goals and (b) results in queries of the form $call((G_1, ..., G_n))$. Calling such a conjunction of goals is implemented in YAP as the following:

```
call((X,Y)) :- !, meta_call(X), call(Y).
call(X) :- meta_call(X).
```

Meta-calls are expensive and repeated failures, backtracks, and re-trials results in repeated re-invocation of this code. There are two things we could to alleviate this: (1) reduce the conjunction of goals to as few as possible; and (2) detect situations where any amount of backtracking over previous goals would not change the result of the computation of successive goals (the two sets of goals are *independent*). These two observations form the basis of the transformations described next.

4 Two Transformations to Reduce Query Costs

4.1 Transformation 1: Eliminate Redundant Literals

We are concerned here with clauses of the form $Head : -Body$ where $Body$ is conjunction of goals $G_1, ..., G_n$ (in the Prolog sense, or literals in the more traditional logical sense. Each of the G_i are calls to user-defined or built-in predicates). From the previous section it is evident that there is at least one meta-call for each element of the conjunction. In general, let N be an upper-bound on the number of solutions to any of the G_i. It follows straightforwardly that in the worst-case, the number of meta-calls is $O(N^n)$. Lowering N requires a re-appraisal of predicate definitions used by clauses (either by re-encoding of some predicates, or by removing them entirely). We do not pursue this here, and concentrate instead on lowering the value of n.

A straightforward reduction in the number of goals being tested (thus lowering n) is achieved by eliminating obviously redundant ones. A simple logical check for this during the execution of the steps described in Fig. 1 is as follows. On iteration i of the procedure in Fig. 1, let P denote $B \cup H_{i-1}$. Let C be a clause being examined in a search. Then, a literal l in C is redundant iff $P \cup \{C\} \equiv P \cup \{C'\}$ where $C' = C - \{l\}$. From this, it is easy to show that l is redundant iff $P \cup \{C\} \models \{C'\}$. This constraint, while providing an exact test for literal-redundancy, is expensive to implement (with a resolution-based theorem prover like that in YAP, it requires checking for the proof of inconsistency from the set $P \cup \{C\} \cup \{\neg C'\}$). Instead, we will employ a simpler test based on the subsumption relation [18]. Recall that C subsumes C' iff there is some substitution θ such that $C\theta \subseteq C'$ and if C subsumes C' then $\{C\} \models \{C'\}$. Clearly therefore if C subsumes C' then $P \cup \{C\} \models \{C'\}$ and l is redundant (the reverse is not true, and hence we call it a "weak" test for redundancy).

The clausal-subsumption test is relatively inexpensive to perform and can result in a reduction of goals being tested. While there is nothing profound about this test, it is not clear from descriptions of existing ILP systems whether they actually employ it during the search[5]. In general, while the efficacy of the transformation will depend on the enumeration operator employed by the search function (that is, whether it is prone to introduction of redundant literals), we expect it to be more useful when the background predicates are non-determinate.

4.2 Transformation 2: Exploit Goal Independence

We observe that when executing a conjunction of goals $G_1, ..., G_n$, failure of a goal G_i will result in attempting to generate more solutions for goals earlier in the sequence. Sometimes this effort is useless, as these solutions would not alter the computation of G_i. There is thus a notion of *goal independence* that we can exploit.

In pure logic programs, goals depend on each other because they share variables. Given a function $vars(T)$ that returns all unbound terms in the term T, two goals G_i and G_j are said to *share*, that is the relation $Shares(G_i, G_j)$ holds, when:

$$i = j \ \vee \ vars(G_i) \cap vars(G_j) \neq \emptyset$$

The definition ensures that the relation *Shares* is reflexive. It is also symmetric and transitive, as all the $=, \cap, \neq, \vee$ relations are both symmetric and transitive. Therefore, *Shares* is an equivalence relation. Given a set of goals $\{G_1, \ldots, G_n\}$, we shall name the equivalence classes established by *Shares* as $I_1, \ldots, I_m$.

Intuitively, we say that a goal G_i in a class I_k is independent of a goal G_j on a class I_l, $k \neq l$, that is, that the computation for G_j can never bind a variable in G_i and therefore that results for G_j can never influence results for G_i in a clause. More precisely, let the conjunction of goals $G = G_1, ..., G_n$ be represented as the set $\{G_1, \ldots, G_n\}$. We claim that (i) if the sets $I_1, \ldots, I_m$ are satisfiable, then so is G; and that (ii) if one of the sets $I_1, \ldots, I_m$ is not satisfiable, G is not satisfiable. Therefore, to prove that G is satisfiable it is sufficient to prove that $I_1, \ldots, I_m$ are satisfiable.

To prove (i) we first notice that if the $I_1, \ldots, I_m$ are satisfiable we must be able to find a solution for each one, say $\sigma_1, \ldots, \sigma_m$. Moreover, from the *Shares* relation we know that if a variable appears in a σ_k it cannot appear in a $\sigma_l, l \neq k$. We thus can compose $\sigma_1, \ldots, \sigma_m$ and still obtain a valid substitution $\sigma = \sigma_1, \ldots, \sigma_m$.

Next, consider a goal G_i. The goal must belong to an equivalence class for *Shares*, that is $\exists k, G_i \in I_k$. Without loss of generality, we can reorder this sets so that $k = 1$. We next show by finite induction on $l \geq m$ that $G_i\sigma = G_i\sigma_1$, and therefore that $G_k\sigma$ is satisfiable:

[5] Programs like WARMR [6] perform extensive checks to ensure that clauses subsumed by others that have previously been shown to be "useless" are not examined further. This is more elaborate than the proposal here, which simply checks for redundant literals within a clause.

- $G_i\sigma_1$ is of course $G_i\sigma_1$, and,
- If $G_i\sigma_1 \ldots \sigma_{l-1} = G_i\sigma_1$, then $G_i\sigma_1 \ldots \sigma_{l-1}\sigma_l = G_i\sigma_1\sigma_l$. But, none of the right-hand applies to a variable in $G_i\sigma_l$ so we have $G_i\sigma_1 \ldots \sigma_l = G_i\sigma_1$.

Thus, for every goal G_i, $G_i\sigma$ and so G_i are satisfiable. Therefore, all goals G_i are satisfiable, hence the conjunction G is satisfiable. To prove (ii) it is sufficient to notice that if a subset of a conjunction is not satisfiable, then the conjunction may not be satisfied.

Checking whether goals are independent is expensive. Ideally, we would like to do so only once when we compile a newly induced clause, not for every time we run the clause against an example. This means we will need to approximate the *Shares* predicate by more general predicates. One solution is to replace $vars(G_i)$ and $vars(G_j)$ by $\alpha(vars(G_i))$ and $\alpha(vars(G_j))$, where α is such that $V \subset \alpha(V)$. We thus ensure the approximation relation or *abstraction* will also be an equivalence relation and will have a set of equivalence classes I's. Moreover, from the definition of α it follows that if $G_i \in I_l \wedge G_j \in I_l$, then there is an I'_m for the abstraction such that $G_i \in I'_m \wedge G_j \in I'_m$. Intuitively, the point is that this abstraction will preserve dependence, although independent goals may not be recognised.

We will perform the transformation while processing a new clause of the form $Head : -G_1, \ldots, G_n$. Our goal is to find the (approximated) equivalence classes in the $G_1, \ldots, G_n$ and generate code that executes them independently.

Our initial observation is is that if at compile-time variables in literal G form the set $V(G)$, then at run time we will have $vars(G) = vars'(V(G))$, where $vars'$ domain is a set of terms instead of on single term. For the sake of simplicity we next abuse terminology somewhat and instead of writing our use of the abstraction function as $\alpha(vars(G))$ we will just use an $\alpha(V(G))$. Next, we base our first abstraction on the idea that at compile-time variables may be in one of two types:

- *Existentially quantified* variables are variables that appear in at least one of the literals G_i but that do not appear in $Head$. For an existentially quantified variable E we will name the set of run-time values it can take as $\mathcal{E} = \{E', E'', E''', \ldots\}$. We know beforehand that for two different variables E_1 and E_2, $\mathcal{E}_1 \neq \mathcal{E}_2$.
- All other variables are assumed to be *universally quantified*. Universally quantified variables appear in the head of clause, and it is conservative to assume that at run-time they were bound to each other before the body of the clause was called.
  We name the set of run-time variables that may appear in a (compile-time) universally quantified variable U_i as $\mathcal{U}_i$. We do not know whether two universally quantified variables U_i and U_j are bound together, so we must assume that $\mathcal{U}_i \cap \mathcal{U}_j \neq \emptyset$. On the other hand, we do know that a new copy of all existentially quantified variables is created when we enter the body of a clause, so $\mathcal{U}_i \cap \mathcal{E}_j = \emptyset$. We name the union of all sets $\mathcal{U}_i$ as $\mathcal{U}$.

Our first abstraction α' summarises the previous observations:

- $\alpha'(\emptyset) = \emptyset$;
- $\alpha'(\{E_i\}) = \mathcal{E}_i$;
- $\alpha'(\{U_i\}) = \mathcal{U}$; and
- $\alpha'(x \cup y) = \alpha'(x) \cup \alpha'(y)$.

Unfortunately, this first abstraction is too general: all goals that include an universally quantified variable or that share variables with one such goal will be considered dependent. Better results stem from adding a new category of compile-time variables:

- Ground (or closed) variables are universally quantified variables that are known to be bound at run-time to ground (or closed) terms. This information may stem from user declarations or from other analysis. Clearly, the set of run-time variables for a ground variable G_i is $\emptyset$.

Our next abstraction α uses the class G:

- $\alpha(\emptyset) = \emptyset$;
- $\alpha(\{E_i\}) = \mathcal{E}_i$;
- $\alpha(\{U_i\}) = \mathcal{U}$;
- $\alpha(\{G_i\}) = \emptyset$; and
- $\alpha(x \cup y) = \alpha(x) \cup \alpha(y)$.

This is sufficient for our purposes (better approximations may be obtainable from the literature on independent and-parallelism [3]). To take advantage of this transformation it is sufficient to implement the following algorithm:

1. Given the original clause:

   $$H \texttt{:-}\ G_1,\ \ldots,\ G_i,\ \ldots,\ G_n.$$

   classify all variables in $G_1, \ldots, G_n$ and calculate the equivalence classes for the approximated sharing relation.
2. Number the literals according to the equivalence class they belong to:

   $$H \texttt{:-}\ G_{1j},\ \ldots,\ G_{ik},\ \ldots,\ G_{nl}.$$

3. Reorder the literals in the clause according to the class they belong to:

   $$H \texttt{:-}\ G_{a1},\ \ldots,\ G_{b1}, \ldots,\ G_{cm},\ \ldots,\ G_{dm}.$$

   The transformation does not introduce or remove goals, so according to the switching lemma it is correct for pure programs.
4. We are interested in any solution, if one exists. Thus we need to compute every class once and if a class has no solution, the computation should wholly fail. The following program transformation uses `cut` to guarantee such a computation:

   $$H \texttt{:-}\ G_{a1},\ \ldots,\ G_{b1}, \texttt{!}, \ldots, \texttt{!},\ G_{cm},\ \ldots,\ G_{dm}.$$

As with the previous transformation, we expect goal independence to be useful when the predicates involved are non-determinate.

5 Empirical Evaluation

The transformations of the previous section are evaluated on four real-world tasks previously examined by ILP systems, namely biochemical problems of predicting mutagenesis and carcinogenesis; an engineering problem of deciding on optimal mesh partitions; and a natural language task of classifying word-tokens (or "tagging" the tokens) in sentences taken from a standard corpus. For each task we compare the ratio of times for theory construction obtained with and without the transformations. The tasks selected are such that two of them (the biochemical ones) consist of highly non-determinate background predicates and the others, of largely determinate predicates. Given that the transformations we have proposed are better suited to problems with non-determinate predicates, our prior expectation is thus:

> Benefits, if any, from the transformations will be observed in the biochemical tasks.

5.1 Domains and Data

The tasks selected are well-known within the ILP literature. Space restrictions prevent us from presenting the problems here. Summary descriptions can be found in [21] and [22]. For complete details, we refer the reader to: [10, 26] (mutagenesis); [23] (carcinogenesis); [7] (mesh); and [4] (tag). All data used here are available electronically by contacting the second author.

5.2 Algorithms and Machines

All experiments use the ILP program Aleph (Version 2) and the Prolog compiler YAP (Version 4.3.1). The current version of Aleph is available at: `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph`, and that of YAP at: `http://www.ncc.up.pt/~vsc/Yap/`.

In the experiments we distinguish between the following: (1) Aleph, to denote Aleph performing a restricted breadth-first branch-and-bound search, without the transformations proposed here, and with an evaluation function that assigns a clause a score that is the difference between the positive and negative examples derivable by using the clause. The number of clauses explored in any search is limited to 100,000; (2) Aleph+R, to denote Aleph with the first transformation incorporated (redundancy check); and (3) Aleph+R+I, to denote Aleph with both transformations incorporated (redundancy check, followed by independence check).

Note that we do not explore the following options: (a) Aleph with independence check only; and (b) Aleph with independence check followed by the redundancy check. Both these are of less interest, as the presence of redundant literals is always undesirable and is best corrected first. All experiments were performed on a SPARC Ultra 30 equipped with a 300MHz processor and 256 megabytes of random access memory.

5.3 Method

We adopt the following method:

For each task (Mutagenesis, Carcinogenesis, Mesh, and Tag)

- Restrict the hypothesis language $\mathcal{L}$ to clauses with at most $3, 4, 5, 6$ literals.
- For each instance of $\mathcal{L}$, construct theories consistent with the data using Aleph, Aleph+R, and Aleph+R+I. Record the time taken to construct the theory in each case $(T_{Aleph}, T_{Aleph+R}, T_{Aleph+R+I})$.
- Estimate the transformation efficacies from ratios of times recorded.

The following details are relevant: (1) Increasing the number of literals allowed in clauses usually increases the size of the search space explored. We are interested in examining the effect of the transformations as this size increases; (2) In each case, we obtain the time taken by the ILP algorithm to construct a theory that is consistent with the data. We recognise of course that in practice, theories that are to some degree inconsistent with the data may be acceptable. We are avoiding these here only to prevent confounding the study with secondary effects that may arise from a comparison of theories with incomparable accuracies; and (3) In every case, the theories constructed by Aleph, Aleph+R, and Aleph+R+I should be identical. This follows from the fact that the transformations proposed are admissible. This has also been verified empirically with the theories obtained.

5.4 Results and Discussion

Figure 3 tabulates the ratio of times for theory construction using the method just described. In the figure, ratios less than 1 indicate an improvement over Aleph without any transformations (percentage gains or losses from a ratio r is

Max. Clause Length	Time Ratios							
	MUT		CANC		MESH		TAG	
	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)
3	1.03	1.17	1.05	1.13	0.98	1.02	1.15	1.03
4	1.00	1.16	0.93	0.92	1.00	1.07	1.19	1.01
5	0.81	0.86	0.51	0.49	1.00	1.09	1.17	1.14
6	0.77	0.77	0.25	0.21	1.02	1.13	1.24	1.31

Fig. 3. Ratio of times for theory construction. All ratios are against the time for theory construction using Aleph without transformations. MUT refers to the mutagenesis task, CANC to carcinogenesis, MESH to mesh-design, and TAG to tagging. The ratio (A) is with the check for redundancy (that is, it refers to $T_{Aleph+R}/T_{Aleph}$). The ratio (B) is with the check for redundancy and goal independence (that is, it refers to $T_{Aleph+R+I}/T_{Aleph}$).

simply $|r-1| \times 100$). The principal details in Fig. 3 are these: (1) The transformations are effective for large clause lengths on the biochemical tasks, and ineffective on the mesh-design and tagging tasks; and (2) When effective, the transformations work better when the sizes of the space to be searched increases.

The ratios reported in Fig. 3 do not account for the fact that in practice, the transformations are applied in sequence. It is instructive to consider the sequential changes in times for theory construction. This is shown in Fig. 4. As before, percentage gains or losses from a ratio r is $|r-1| \times 100$.

Max. Clause Length	Time Ratios							
	MUT		CANC		MESH		TAG	
	(A)	(B)	(A)	(B)	(A)	(B)	(A)	(B)
3	1.03	1.14	1.05	1.08	0.98	1.04	1.15	0.90
4	1.00	1.16	0.93	0.98	1.00	1.06	1.19	0.85
5	0.81	1.05	0.51	0.95	1.00	1.08	1.17	0.97
6	0.77	1.00	0.25	0.82	1.02	1.10	1.24	1.05

Fig. 4. Effect of applying the transformations in sequence. The ratio (A) refers to $T_{Aleph+R}/T_{Aleph}$ and is thus identical to the column (A) in Fig. 3. The ratio B refers to $T_{Aleph+R+I}/T_{Aleph+R}$ and measures the effect of performing the independence check *after* the check for redundancy. Thus the entries $0.81, 1.05$ under MUT (for length $= 5$) are to be interpreted as follows. The redundancy check resulted in a 19% speed-up. Adding the independence check resulted in a 5% slow-down after this.

These results appear to confirm our prior expectation, namely that benefits, if any, will manifest themselves in the biochemical tasks. They also provide evidence for some other observations:

- If theories required are known to consist fairly short clauses (less than 4 literals), then there appears to be little to gain from these transformations.
- Predicates used by mesh and tag problems are almost entirely functional and the transformations act as overheads that provide no benefit. On the other hand, the greater the non-determinacy of background predicates, the greater appear to be the gains. This is illustrated by the difference in entries between the two biochemical tasks. The non-determinacy of background predicates for the carcinogenesis task is significantly greater than those for mutagenesis. For example, the `atm/5` appears in both problems, with the same meaning: given a compound name, it returns a description of the atoms (in a chemical sense) in that compound. In carcinogenesis, for a given compound, this predicate can succeed up to 214 times. In contrast, this number is just 40 for mutagenesis. As greater non-determinacy usually increases the theorem-proving effort required—especially when the lengths of clauses is high—benefits from the transformations proposed is accentuated correspondingly.

– The entries under column (B) in Fig. 4 suggest that as clause length increases, the independence check appears to be increasingly effective for the non-determinate domains and increasingly ineffective for the determinate ones. The former trend persists in further experiments: systematically increasing clause lengths to 10 see the entries in column (B) decrease to 0.97 (MUT) and 0.46 (CANC). No such consistent picture emerges with the determinate domains (that is, increasing clause lengths does not lead to a steady increase in the corresponding ratio).

These observations lead us to the following prescription:

– If background predicates are largely functional, or theories are required to contain only very short clauses (4 literals or less), then do not use these transformations;
– If background predicates are non-determinate and theories require clauses of moderate length, then only use the transformation that eliminates redundant literals;
– If background predicates are non-determinate and theories require long clauses, then use both transformations.

6 Concluding Remarks

As ILP systems move from the realm of research to one of technology, implementation issues and concerns of efficiency become increasingly important. Here we have described two simple clause-transformation techniques that are directed towards reducing the theorem-proving effort that is at the heart of many ILP systems. Empirical evaluation suggests that the transformations can provide efficiency gains for problems that require complex theories that use highly non-determinate background predicates. It is of interest to see if these results are replicated on other datasets with similar characteristics, and with other ILP systems that use possibly different enumeration strategies.

The approach adopted in this paper is in the long tradition source-to-source program transformations: changes at the source-level that can improve efficiency without altering correctness [13]. While the two suggestions here by no means exhaust the transformations of this type, it is only one side of the story. Efficiency gains are as much to be made by tailoring compilers of logic programs to account for what ILP systems do. In concluding his empirical study, Knuth states: "...the design of compilers should be influenced by what programmers want to do. An alternative point of view is that programmers should be strongly influenced by what their compilers do; a compiler writer in his infinite wisdom may in fact know what is really good for the programmer and would like to steer him towards a proper course. This viewpoint has some merit, although it has often been carried to extremes in which programmers have to work harder and make unnatural constructions just so the compiler writer has an easier job." Our experience suggests that compiler writers for logic programs do not subscribe to the alternative point of view.

Acknowledgments

This work has been partly supported by Fundação da Ciência e Tecnologia under the project Dolphin (PRAXIS/2 /2.1/TIT/1577/95), and by the PROTEM5 CNPq-NSF Collaboration Project $CLoP^n$. A.S is supported by a Nuffield Trust Fellowship at Green College, Oxford.

References

[1] R. Benigni. (Q)SAR prediction of chemical carcinogenicity and the biological side of the structure activity relationship. In *Proceedings of The Eighth International Workshop on QSARs in the Environmental Sciences*, 1998. Held in Baltimore, May 16–20, 1998.

[2] I. Bratko and M.Grobelnik. Inductive learning applied to program construction and verification. In *Third International Workshop on Inductive Logic Programming*, pages 279–292, 1993. Available as Technical Report IJS-DP-6707, J. Stefan Inst., Ljubljana, Slovenia.

[3] M. Codish, M. Bruynooghe, M. G. de la Banda, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming*, 32(3), 1997.

[4] J. Cussens. Part-of-Speech Tagging Using Progol. In S. Džeroski and N. Lavrač, editors, *Proceedings of the Seventh International Workshop on ILP*, volume 1297 of *LNAI*, pages 93–108. Springer, 1997.

[5] A.K. Debnath, R.L Lopez de Compadre, G. Debnath, A.J. Schusterman, and C. Hansch. Structure-Activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786 – 797, 1991.

[6] L. Dehaspe, H. Toivonen, and R.D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD-98)*, pages 30–36. AAAI Press, 1998.

[7] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive Logic Programming*, pages 453–472. Academic Press, London, 1992.

[8] S. Dzeroski, L. Dehaspe, B. Ruck, and W. Walley. Classification of river water quality data using machine learning. In *Proceedings of the Fifth International Conference on the Development and Application of Computer Techniques Environmental Studies*, 1994.

[9] C. Feng. Inducing temporal fault dignostic rules from a qualitative model. In S. Muggleton, editor, *Inductive Logic Programming*, pages 473–486. Academic Press, London, 1992.

[10] R.D. King, S.H. Muggleton, A. Srinivasan, and M.J.E. Sternberg. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proc. of the National Academy of Sciences*, 93:438–442, 1996.

[11] R.D. King, S.H. Muggleton, and M.J.E. Sternberg. Drug design by machine learning: The use of inductive logic programming to model the structure-activity relationships of trimethoprim analogues binding to dihydrofolate reductase. *Proc. of the National Academy of Sciences*, 89(23):11322–11326, 1992.

[12] D. E. Knuth. An Empirical Study of FORTRAN Programs. *Software—Practice and Experience*, 1:105–133, 1971.
[13] D. B. Loveman. Program improvement by source-to-source transformation. *JACM*, 24(1):121–145, 1977.
[14] S. Muggleton. Inductive Logic Programming: derivations, successes and shortcomings. *SIGART Bulletin*, 5(1):5–11, 1994.
[15] S. Muggleton. Inverse Entailment and Progol. *New Gen. Comput.*, 13:245–286, 1995.
[16] S. Muggleton, R. King, and M. Sternberg. Predicting protein secondary structure using inductive logic programming. *Protein Engineering*, 5:647–657, 1992.
[17] S.H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, Tokyo, 1990. Ohmsha.
[18] S. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer, Berlin, 1997.
[19] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
[20] M. Sebag and C. Rouveirol. Tractable Induction and Classification in First-Order Logic via Stochastic Matching. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann, Los Angeles, CA, 1997.
[21] A. Srinivasan. A study of two probabilistic methods for searching large spaces with ILP. *Data Mining and Knowledge Discovery (under review)*, 1999.
[22] A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.
[23] A. Srinivasan and R.D. King. Carcinogenesis predictions using ILP. In N. Lavrac and S. Dzeroski, editors, *Proceedings of the Seventh International Workshop on Inductive Logic Programming (ILP97)*, volume 1297 of *LNAI*, pages 273–287, Berlin, 1997. Springer. A version also in Intelligent Data Analysis in Medicine, Kluwer.
[24] A. Srinivasan, R.D. King, and D.W. Bristol. An assessment of submissions made to the Predictive Toxicology Evaluation Challenge. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence (IJCAI-99)*. Morgan Kaufmann, Los Angeles, CA, 1999.
[25] A. Srinivasan, R.D. King, S.H. Muggleton, and M.J.E. Sternberg. The Predictive Toxicology Evaluation Challenge. In *Proceedings of the Fifteenth International Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann, Los Angeles, CA, 1997.
[26] A. Srinivasan, S.H. Muggleton, R.D. King, and M.J.E. Sternberg. Theories for mutagenicity: a study of first-order and feature based induction. *Artificial Intelligence*, 85:277–299, 1996.
[27] J. Zelle and R. Mooney. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 817–822. Morgan Kaufmann, 1993.

A Prolog Code for the Transformations

The following Prolog code assumes the definitions of predicates for manipulating ordered sets. These are available within the library modules distributed with YAP.

```
% Transformation 1
remove_redundant((Head:-Body),(Head1:-Body1)):-
        goals_to_list((Head,Body),ClauseL),
        remove_subsumed(ClauseL,[Head1|Body1L]),
        (Body1L = [] -> Body1 = true; list_to_goals(Body1L,Body1)).

% Transformation 2
reorder_clause((Head:-Body), Clause) :-
        term_variables(Head,LHead),
        number_goals_and_get_vars(Body,LHead,1,_,[],Conj),
        calculate_independent_sets(Conj,[],BSets),
        compile_clause(BSets,Head,Clause).

number_goals_and_get_vars((G,Body),LHead,I0,IF,L0,[g(I0,LVF,NG)|LGs]):-!,
        I is I0+1,
        get_goal_vars(G,LHead,LVF,NG),
        number_goals_and_get_vars(Body,LHead,I,IF,L0,LGs).
number_goals_and_get_vars(G,LHead,I,I,L0,[g(I,LVF,NG)|L0]) :-
        get_goal_vars(G,LHead,LVF,NG).

get_goal_vars(G,LHead,LVF,G) :-
        term_variables(G,LV0),
        sort(LV0,LVI),
        ord_subtract(LVI,LHead,LVF).

calculate_independent_sets([],BSets,BSets).
calculate_independent_sets([G|Ls],BSets0,BSetsF) :-
        add_goal_to_set(G,BSets0,BSetsI),
        calculate_independent_sets(Ls,BSetsI,BSetsF).

add_goal_to_set(g(I,LV,G),Sets0,SetsF) :-
        add_to_sets(Sets0,LV,[g(I,LV,G)],SetsF).

add_to_sets([],LV,Gs,[[LV|Gs]]).
add_to_sets([[LV|Gs]|Sets0],LVC,GsC,[[LV|Gs]|SetsF]) :-
        ord_disjoint(LV,LVC), !,
        add_to_sets(Sets0,LVC,GsC,SetsF).
add_to_sets([[LV|Gs]|Sets0],LVC,GsC,SetsF) :-
        ord_union(LV,LVC,LVN),
        join_goals(Gs,GsC,GsN),
        add_to_sets(Sets0,LVN,GsN,SetsF).

join_goals([],L,L):- !.
join_goals(L,[],L):- !.
join_goals([g(I1,VL1,G1)|T],[g(I2,VL2,G2)|T2],Z) :-
        I1 < I2, !,
        Z = [g(I1,VL1,G1)|TN],
        join_goals(T,[g(I2,VL2,G2)|T2],TN).
```

```
join_goals([H|T],[g(I2,VL2,G2)|T2],Z) :-
        Z = [g(I2,VL2,G2)|TN],
        join_goals(T,[H|T2],TN).

compile_clause(Goals,Head,(Head:-Body)):-
        compile_clause2(Goals,Body).

compile_clause2([[_|B]], B1):-
        !,
        glist_to_goals(B,B1).
compile_clause2([[_|B]|Bs],(B1,!,NB)):-
        glist_to_goals(B,B1),
        compile_clause2(Bs,NB).

% remove literals subsumed in the body of a clause
remove_subsumed([Head|Lits],Lits1):-
        delete(Lit,Lits,Left),
        \+(\+(redundant(Lit,[Head|Lits],[Head|Left]))), !,
        remove_subsumed([Head|Left],Lits1).
remove_subsumed(L,L).

% determine if Lit is subsumed by a body literal
% this is not a very efficient implementation
redundant(Lit,Lits,[Head|Body]):-
        copy_term([Head|Body],Rest1),
        member(Lit1,Body),
        Lit = Lit1,
        subsumes(Lits,Rest1).

subsumes(Lits,Lits1):-
        \+(\+((numbervars(Lits,0,_),numbervars(Lits1,0,_),
                                                subset1(Lits,Lits1)))).

% General utilities
list_to_goals([Goal],Goal):- !.
list_to_goals([Goal|Goals],(Goal,Goals1)):-list_to_goals(Goals,Goals1).

glist_to_goals([g(_,_,Goal)],Goal):- !.
glist_to_goals([g(_,_,Goal)|Goals],(Goal,Goals1)):-
                                            glist_to_goals(Goals,Goals1).

goals_to_list((true,Goals),T):- !, goals_to_list(Goals,T).
goals_to_list((Goal,Goals),[Goal|T]):- !, goals_to_list(Goals,T).
goals_to_list(true,[]):- !.
goals_to_list(Goal,[Goal]).

subset1([],_).
subset1([Elem|Elems],S):- member1(Elem,S), !, subset1(Elems,S).
```

```
member1(H,[H|_]):- !.
member1(H,[_|T]):- member1(H,T).

member(H,[H|_]).
member(H,[_|T]):- member(H,T).

delete(H,[H|T],T).
delete(H,[H1|T],[H1|T1]):- delete(H,T,T1).
```

Searching the Subsumption Lattice by a Genetic Algorithm

Alireza Tamaddoni-Nezhad and Stephen H. Muggleton

Department of Computer Science
University of York, York, YO1 5DD, UK
{alireza,stephen}@cs.york.ac.uk

Abstract. A framework for combining Genetic Algorithms with ILP methods is introduced and a novel binary representation and relevant genetic operators are discussed. It is shown that the proposed representation encodes a subsumption lattice in a complete and compact way. It is also shown that the proposed genetic operators are meaningful and can be interpreted in ILP terms such as lgg(least general generalization) and mgi(most general instance). These operators can be used to explore a subsumption lattice efficiently by doing binary operations (e.g. and/or). An implementation of the proposed framework is used to combine Inverse Entailment of CProgol with a genetic search.

1 Introduction

Using complete and efficient methods for searching the refinement space of hypothesis is a challenging issue in current ILP systems. Different kinds of greedy methods as well as heuristics (e.g. information gain) have been successfully employed to cope with complexity of search for inducing first-order concepts from examples. However, more powerful heuristics are required for inducing complex concepts and for searching very large search spaces. Genetic Algorithms (GAs) have great potential for this purpose. GAs are multi-point search methods (and less sensitive to local optima) which can search through a large space of symbolic as well as numerical data. Moreover, because of their robustness and adaptive characteristics, GAs are suitable methods for optimization and learning in many real world applications [6, 4]. In terms of implementation, GAs are highly parallel and can be easily implemented in parallel and/or distributed environments [4]. However, GAs are syntactically restricted and cannot represent a priori knowledge that already exists about the domain. On the other hand, ILP is a well known paradigm that benefits from the expressive power inherited from logic and logic programming [12]. Hence, it is likely that a combination of ILP and GAs can overcome the limitation of each individual method and can be used to cope with some complexities of real-world applications.

Even though GAs have been used widely for optimization and learning in many domains, a few genetic-based systems in first-order domain already exist. Some of these systems [3, 2, 5] follow conventional genetic algorithms and represent problem solutions by fixed length bit-strings. Other systems [16, 10, 7]

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 243–252, 2000.

use a hierarchical representation and evolve a population of logic programs in a Genetic Programming (GP) [9] manner. These genetic-based systems confirm that genetic algorithms and evolutionary computing can be interesting alternatives for learning first-order concepts from examples. However, in most of these systems genetic algorithm is the main learning mechanism and therefore they cannot benefit from background knowledge during the learning process.

This paper aims to present a framework for combining GAs with ILP methods by introducing a novel binary encoding and relevant genetic operators. In this framework, unlike other genetic-based systems, representation and operators can be interpreted in well known ILP terms such as θ-subsumption, lgg (least general generalization) and mgi(most general instance) [14, 15]. This representation and its ILP interpretation are introduced in the next section. Genetic operators and their relationship with ILP concepts are examined in section 3. Section 4 describes an implementation of the proposed framework for combining Inverse Entailment in CProgol [13] with a genetic algorithm. Evaluations and related works are discussed in section 5. Finally, section 6 summarizes the results and concludes this paper.

2 Representation and Encoding

Every application of GAs requires formulating problem solutions in such a way that they can be processed by genetic operators. According to *the principle of minimal alphabet* in GAs [4], a binary representation could be the best coding for representing problem solutions in GAs. The lack of a proper binary representation (and consequently difficulties for definition and implementation of genetic operators) has been the main problem for applying GAs in first-order domain.

In this section we present a binary representation which encodes the refinement space of a clause. Consider clause C with n variable occurrences in head and body. The relationships between these n variable occurrences can be represented by a graph having n vertices in which there exists an edge between vertices v_i and v_j if ith and jth variable occurrences in the clause represent the same variable. For example variable binding in clause *p(A,B):-q(A,C),r(C,B)* can be represented by an undirected graph as shown in Figure 1, this clause also can be represented by the binary matrix shown in the figure. In this matrix entry m_{ij} is 1 if ith and jth variable occurrences in the clause represent the same variable and m_{ij} is 0 otherwise. We show that this simple representation has interesting properties for searching the subsumption lattice bounded below by clause C. First, we need to show the mappings between clauses and binary matrices.

Definition 1 (Binding Set). *Let B and C both be clauses. C is in binding set $\mathcal{B}(B)$ if there exists a variable substitution*[1] *θ such that $C\theta = B$.*

In Definition 1, the variables in B induce a set of equivalence classes over the variables in any clause $C \in \mathcal{B}(B)$. Thus we could write the equivalence class of

[1] substitution $\theta = \{v_i/u_j\}$ is a variable substitution if all v_i and u_j are variables.

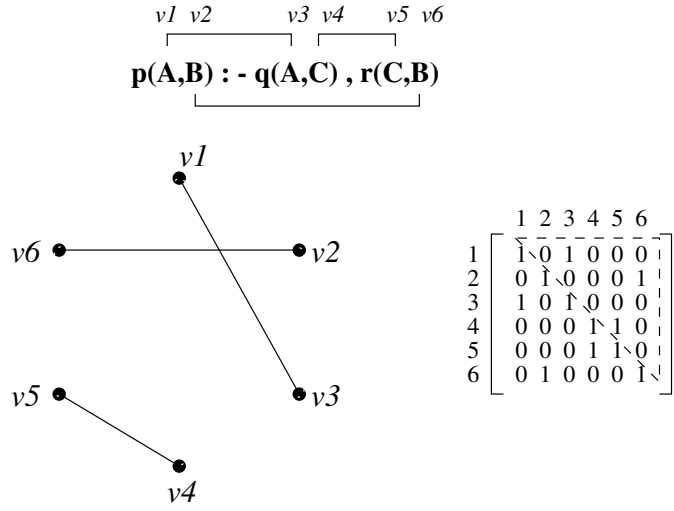


Fig. 1. Binding Graph and Binding Matrix for clause *p(A,B):-q(A,C),r(C,B).*

u in variable substitution θ as $[v]_u$, the set of all variables in C such that v/u is in θ. We define a binary matrix which represents whether variables v_i and v_j are in the same equivalence class or not.

Definition 2 (Binding Matrix). *Suppose B and C are both clauses and there exists a variable substitution θ such that $C\theta = B$. Let C have n variable occurrences in head and body representing variables $\langle v_1, v_2, \ldots, v_n \rangle$. The binding matrix of C is an $n \times n$ matrix M in which m_{ij} is 1 if there exist variables v_i, v_j and u such that v_i/u and v_j/u are in θ and m_{ij} is 0 otherwise. We write $M(v_i, v_j) = 1$ if $m_{ij} = 1$ and $M(v_i, v_j) = 0$ if $m_{ij} = 0$.*

Definition 3. *Let M be an $n \times n$ binary matrix. M is in the set of normalized binding matrices $\mathcal{M}_n$ if M is symmetric and for each $1 \leq i \leq n$, $1 \leq j \leq n$ and $1 \leq k \leq n$, $m_{ij} = 1$ if $m_{ik} = 1$ and $m_{kj} = 1$.*

Definition 4. *The mapping function $M : \mathcal{B}(B) \rightarrow \mathcal{M}_n$ is defined as follows. Given clause $C \in \mathcal{B}(B)$ with n variable occurrences in head and body representing variables $\langle v_1, v_2, \ldots, v_n \rangle$, $M(C)$ is an $n \times n$ binary matrix in which m_{ij} is 1 if variables v_i and v_j are identical and m_{ij} is 0 otherwise.*

Definition 5. *The mapping function $C : \mathcal{M}_n \rightarrow \mathcal{B}(B)$ is defined as follows. Given a normalized $n \times n$ binding matrix M, $C(M)$ is a clause in $\mathcal{B}(B)$ with n variable occurrences $\langle v_1, v_2, \ldots, v_n \rangle$, in which variables v_i and v_j are identical if m_{ij} is 1.*

Definition 6. *Let P and Q be in $\mathcal{M}_n$. It is said that $P \subseteq Q$ if for each entry $p_{ij} \in P$ and $q_{ij} \in Q$, p_{ij} is 1 if q_{ij} is 1. $P = Q$ if $P \subseteq Q$ and $Q \subseteq P$. $P \subset Q$ if $P \subseteq Q$ and $P \neq Q$.*

Definition 7. *Clause C subsumes clause D, $C \succeq D$ if there exists a substitution θ such that $C\theta \subseteq D$ (i.e. every literal in $C\theta$ is also in D). C properly subsumes D, $C \succ D$ if $C \succeq D$ and $D \not\succeq C$.*

In Definition 7, θ can be every substitution, however, in this paper we assume that θ is a variable substitution. The following theorems represent the relationship between binary matrices and the subsumption of clauses.

Lemma 1. *For each M_1 and M_2 in $\mathcal{M}_n$, if $C(M_1) \succ C(M_2)$ and there does not exist clause C' such that $C(M_1) \succ C' \succ C(M_2)$ then there exists unique $\langle v_i, v_j \rangle$, $i < j$ such that $M_1(v_i, v_j) = 0$ and $M_2(v_i, v_j) = 1$ and for each u' and v', $M_1(u', v') = M_2(u', v')$ if $\langle u', v' \rangle \neq \langle v_i, v_j \rangle$.*

Proof. Suppose $C(M_1) \succ C(M_2)$ and there does not exist clause C' such that $C(M_1) \succ C' \succ C(M_2)$. Therefore there exist variables v_i and v_j, $i < j$ such that $C(P_1)\{v_i/v_j\} = C(P_2)$. According to Definition 2 it must be the case that $M_1(v_i, v_j) = 0$ and $M_2(v_i, v_j) = 1$ and $M_1(u', v') = M_2(u', v')$ if $\langle u', v' \rangle \neq \langle v_i, v_j \rangle$. □

Theorem 1. *For each M_1 and M_2 in $\mathcal{M}_n$ if $C(M_1) \succ C(M_2)$ and there does not exist clause C' such that $C(M_1) \succ C' \succ C(M_2)$ then $M_1 \subset M_2$.*

Proof. Suppose $M_1 \not\subset M_2$, thus there exists u' and v' such that $M_1(u', v') = 1$ and $M_2(u', v') = 0$. But according to Lemma 1 there exists unique $\langle v_i, v_j \rangle$, $i < j$ such that $M_1(v_i, v_j) = 0$ and $M_2(v_i, v_j) = 1$ and for each u' and v', $M_1(u', v') = M_2(u', v')$ if $\langle u', v' \rangle \neq \langle v_i, v_j \rangle$. This contradicts the assumption and completes the proof. □

Theorem 2. *For each clause B and matrices M_1 and M_2 in $\mathcal{M}_n$ such that $C(M_1) \in \mathcal{B}(B)$ and $C(M_2) \in \mathcal{B}(B)$, $C(M_1) \succ C(M_2)$ if and only if $M_1 \subset M_2$.*

Proof. $\Rightarrow$: Either there does not exists clause C' such that $C(M_1) \succ C' \succ C(M_2)$ or there exist clauses C_1 and C_2 and $\ldots C_n$ such that $C(M_1) \succ C_1 \succ C_2 \succ \ldots C_n \succ C(M_2)$. In case 1 theorem holds from Theorem 1, in case 2 theorem holds by transitivity.

$\Leftarrow$: Suppose $M_1 \subset M_2$. Therefore there exist variables v_i and v_j, $i < j$ such that $M_1(v_i, v_j) = 0$ and $M_2(v_i, v_j) = 1$. Then according to Definition 2, $C(P_1)\{v_i/v_j\} = C(P_2)$. Hence, $C(M_1) \succ C(M_2)$. □

Definition 8 (Subsumption Set). *Let B and C both be clauses. C is in subsumption set $\mathcal{S}(B)$ if $C \in \mathcal{B}(B)$ and $M(C) \subset M(B)$.*

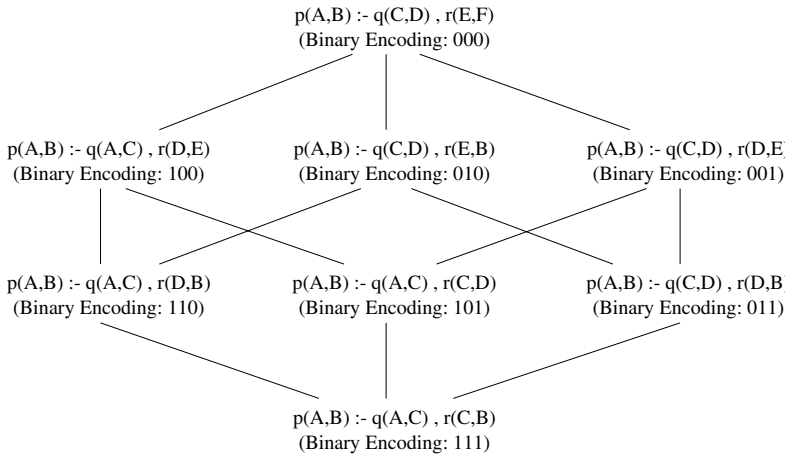


Fig. 2. Subsumption lattice bounded below by clause *p(A,B):-q(A,C),r(C,B)* and relevant binary encodings.

Theorem 3. *For each clause C and B such that $C \in \mathcal{S}(B)$, $C \succ B$.*

Proof. Suppose $C \in \mathcal{S}(B)$, then $M(C) \subset M(B)$. But according to Theorem 2 $C \succ B$. □

A binding matrix is a symmetric matrix in which diagonal entries are 1. In practice, we only maintain entries in top (or down) triangle of the matrix. Furthermore, in our implementation (see section 4), we are interested in a subsumption lattice bounded below by a particular clause. According to Theorem 3, $\mathcal{S}(B)$ represents a subsumption lattice bounded below by clause B. Each member of $\mathcal{S}(B)$ can be encoded by a bit-string in which each bit corresponds to a 1 entry of the matrix.

Example 1. Figure 2 shows the subsumption lattice bounded below by the clause $p(A, B) : -q(A, C), r(C, B)$. Entries m_{13}, m_{26} and m_{45} of the binding matrix in Figure 1 are encoded by three bits as shown in Figure 2.

3 Genetic Refinement Operators

After a proper representation, well designed genetic operators are important factors for success of a genetic algorithm. Genetic operators introduce new individuals into population by changing or combining the genotype of best-fit individuals during an evolutionary process. In this section we present three genetic operators which can be interpreted in ILP terms if applied on the binary representation introduced in the previous section. These operators are and-operator, or-operator and one-point crossover. We show that these operators are equivalent to some

of ILP concepts such as *lgg (least general generalization)* and *mgi(most general instance)* [15].

Definition 9. *Let M_1 and M_2 be in $\mathcal{M}_n$. $M = (M_1 \wedge M_2)$ is an $n \times n$ matrix and for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ and $c_{ij} = 1$ and $a_{ij} = 0$ otherwise.*

Similar to and-operator, or-operator $(M_1 \vee M_2)$ is constructed by bitwise OR-ing of M_1 and M_2 entries.

Definition 10. *Let M_1 and M_2 be in $\mathcal{M}_n$. $M = (M_1 \vee M_2)$ is an $n \times n$ matrix and for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$, $a_{ij} = 1$ if $b_{ij} = 1$ or $c_{ij} = 1$ and $a_{ij} = 0$ otherwise.*

Definition 11. *Let M_1 and M_2 be in $\mathcal{M}_n$ and $1 \leq s_1 \leq n$ and $1 \leq s_2 \leq n$ be randomly selected natural numbers. $M = M_1 \otimes M_2$ is an $n \times n$ matrix and for each $a_{ij} \in M$, $b_{ij} \in M_1$ and $c_{ij} \in M_2$ it is the case that $a_{ij} = b_{ij}$ if $i < s_1$ or $i = s_1$ and $j \leq s_2$ and $a_{ij} = c_{ij}$ otherwise.*

In Definition 11 operator $\otimes$ is equivalent to one-point crossover as defined in genetic algorithms' literatures [4]. Now, we show some interesting properties of these simple operators.

Theorem 4. *For each clause B and matrices M_1, M_2 and M in $\mathcal{M}_n$ such that $C(M_1) \in \mathcal{B}(B)$, $C(M_2) \in \mathcal{B}(B)$ and $C(M) \in \mathcal{B}(B)$, $C(M) = lgg(C(M_1), C(M_2))$ if and only if $M = (M_1 \wedge M_2)$.*

Proof. $\Rightarrow$: Suppose $C(M) = lgg(C(M_1), C(M_2))$. According to the definition of lgg, firstly $C(M) \succ C(M_1)$ and $C(M) \succ C(M_2)$ and according to Theorem 2 and Definition 9 $M \subset (M_1 \wedge M_2)$. Secondly for each binding matrix M' if $C(M') \succ C(M_1)$ and $C(M') \succ C(M_2)$ then $C(M') \succ C(M)$, therefore if $M' \subset (M_1 \wedge M_2)$ then $M' \subset M$. Therefore $(M_1 \wedge M_2) \subset M$ and $M \subset (M_1 \wedge M_2)$, hence $M = M_1 \wedge M_2$.

$\Leftarrow$: Suppose $M = M_1 \wedge M_2$. Therefore $M \subset M_1$ and $M \subset M_2$ and according to Theorem 2 $C(M) \succ C(M_1)$ and $C(M) \succ C(M_2)$. Therefore $C(M)$ is a common generalization of $C(M_1)$ and $C(M_2)$. We show that $C(M)$ is the least general generalization of $C(M_1)$ and $C(M_2)$. For each binding matrix M' in $\mathcal{M}_n$ it must be the case that if $C(M') \succ C(M_1)$ and $C(M') \succ C(M_2)$ then $C(M') \succ C(M)$. Suppose $C(M') \not\succ C(M)$ then there exist u and v such that $M'(u, v) = 1$ and $M(u, v) = 0$. If $M'(u, v) = 1$ then $M_1(u, v) = 1$ and $M_2(u, v) = 1$ and this contradicts $M(u, v) = 0$ and completes the proof. □

By a similar proof it can be shown that the or-operator is equivalent to *mgi* [2].

Theorem 5. *For each clause B and matrices M_1, M_2 and M in $\mathcal{M}_n$ such that $C(M_1) \in L(B)$, $C(M_2) \in L(B)$ and $C(M) \in L(B)$, $C(M) = mgi(C(M_1), C(M_2))$ if and only if $M = M_1 \vee M_2$.*

Proof. Symmetric with proof of Theorem 4. □

[2] The result of operator $\otimes$ can be compared with the result of W operator (predicate invention) in ILP. This property is not discussed in this paper.

Example 2. In Figure 2, *lgg* and *mgi* of each pair of clauses can be obtained by AND-ing and OR-ing of their binary strings.

Because these operators randomly change some entries of the binding matrices, it is possible that the resulting matrix be inconsistent with Definition 2. Such a matrix can be normalized by a closure using Definition 3 before mapping it into a clause.

4 Implementation

In our first attempt, we employed the proposed representation to combine Inverse Entailment in CProgol4.4 with a genetic algorithm. In this implementation genetic search is used for searching the subsumption lattice bounded below by the bottom-clause ($\perp$). According to Theorem 3 the search space bounded by the bottom clause can be represented by $\mathcal{S}(\perp)$. We encoded members of $\mathcal{S}(\perp)$ by bit-strings (as described in section 2) and then applied a genetic algorithm to evolve a randomly generated population of clauses in which each individual corresponds to a member of $\mathcal{S}(\perp)$. Because of simple representation and straightforward operators any standard genetic algorithm can be used for this purpose.

We used a variant of *Simple Genetic Algorithm(SGA)* [4] and modified it to suit the representation introduced in section 2. This genetic search is guided by an *evaluation function* which is similar to one used in A^*-like search of Progol but normalized between 0 and 1. Unlike Progol's refinement operators which non-deterministically decide to include (or exclude) each atom of $\perp$ in current hypothesis, genetic refinement operators evolve bindings of the hypothesis in $\mathcal{S}(\perp)$. Atoms which violate the mode declaration language [13], defined by the user, are filtered from each clause before evaluation. The details for Progol's refinement operator, algorithm for building $\perp$ and A^*-like search can be found in [13] and the algorithm and other details of *Simple Genetic Algorithm(SGA)* can be found in [4]. In the first experiment, we applied the genetic search to learn Michlaski's trains problem [11]. Figure 3 compares the performance of the genetic search with a random search in which each population is generated randomly as in the initial generation. In all experiments (10/10) a correct solution was discovered by the genetic search before generation 20 (standard deviations for the average fitness mean over 10 runs are shown as error bars). In this experiment, the following setting was used for *SGA* parameters: $popsize = 30$, $p_c = 0.6$ and $p_m = 0.0333$.

Preliminary results show that the standard Progol A^*-like search exhibits better performance in learning hypothesis with small and medium complexities. However, the performance of genetic search is less dependent on the complexity of hypothesis, whereas A^*-like search shows a great dependency on this factor. Moreover, genetic search can find the correct solution for some special cases which the solution is beyond the exploration power of A^*-like search due to its incompleteness [13, 1].

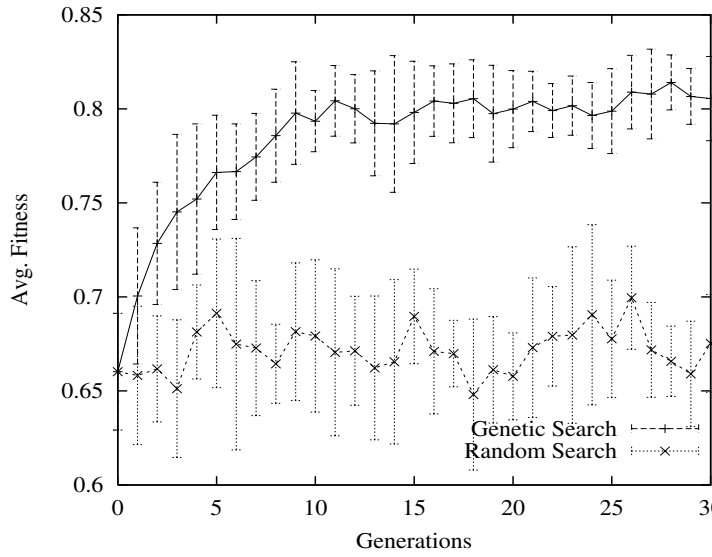


Fig. 3. Convergence of the genetic search in the trains problem.

5 Discussion and Related Works

The actual completeness and complexity exhibited by the standard A^*-like search of Progol depends upon the order of atoms in the bottom clause and upon the distance of the acceptable hypothesis from the root of the search tree. In contrast, because of a multi-point strategy it uses, the genetic algorithm is more regular search method and is not affected by the order of atoms in the bottom clause. Therefore it is reasonable that genetic algorithm is able to find some solutions which are beyond the exploration power of the A^*-like search.

As mentioned earlier, one main difficulty in order to apply GAs in first-order domain concerns formulating first-order hypothesis into bit-strings. GA-SMART [3] and later REGAL [2] and DOGMA [5] were relation learning systems which tackled this problem by restricting concept description language and introducing language templates. A template, in GA-SMART, is a fixed length CNF formula which must be obtained from domain theory (or defined by the user). Mapping a formula into bit-string is done by setting the corresponding bits to represent the occurrences of predicates in the formula. The main problem of this method is that the number of conjuncts in the template grows combinatorially with the number of predicates. In REGAL and DOGMA a template is a conjunction of internally disjunctive predicates which introduce a more complicated language, but still incomplete and poor for representing some features (e.g. continuous attributes). Other systems including GILP [16], GLPS [10] and STEPS [7] use hierarchical representations rather than using fixed length bit-strings. These systems evolve a population of logic programs in a Genetic Pro-

gramming (GP) [9] manner. Most of the above mentioned systems cannot use any form of intensional background knowledge and a few of them use limited forms of background knowledge just for generating initial population or seeding the population.

On the other hand, in our proposed framework, binary representation of hypothesis is compact and complete in comparison to all other methods which use a binary encoding. Encoding of solutions is based on a bottom clause constructed according to the background knowledge using some ILP methods such as Inverse Entailment. Moreover, as shown in section 2 and section 3, the proposed encoding and operators can be interpreted in well known ILP concepts. Hence, in terms of genetic algorithms theory, the proposed framework is not only consistent with *the principal of minimal alphabets* in GAs but also it is along the right lines of *the principle of meaningful building blocks* [4].

6 Conclusions and Further Work

In this paper we have introduced a framework for combining GAs with ILP methods. Simple but complete and compact binary representation for encoding clauses and ILP interpretations of this representation (and relevant operators) are the major novelty of the proposed framework.

An implementation of this framework is used to combine Inverse Entailment of CProgol with a genetic search. Even though this implementation justifies the properness of the proposed framework, it could be improved in many ways. An improvement might be using more sophisticated genetic algorithms (e.g. distributed genetic algorithms) rather than using a simple genetic algorithm. More experiments are also required in complex domains and noisy domains. The ILP interpretation of the proposed genetic operators can be used to guide the genetic search towards the interesting areas of the search space by specialization and/or generalization as it is done in usual ILP systems. As genetic refinement operators introduced in this paper concern specialization as well as generalization, the proposed framework could be useful for theory revision as well.

Undoubtedly, GAs and ILP are on opposite sides in the classification of learning processes [8]. While GAs are known as empirical or BK-poor, ILP could be considered as BK-intensive method in this classification. We conclude that the framework proposed in this paper can be considered as a *bridge* between these two different paradigms to utilize the distinguishable benefits of each method in a hybrid system.

Acknowledgements

Alireza Tamaddoni-Nezhad would like to thank his wife, Parisa Nejabati, for her support during the writing of this paper. Many thanks are also due to the three anonymous referees for their comments, Suresh Manandhar for discussions on the related topics and Alen Varšek for providing us a summary of [16].

References

[1] L. Badea and M. Stanciu. Refinement operators can be (weakly) perfect. In S. Džeroski and P. Flach, editors, *Proceedings of the 9th International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 21–32. Springer-Verlag, 1999.

[2] A. Giordana and F. Neri. Search-intensive concept induction. *Evolutionary Computation Journal*, 3(4):375–416, 1996.

[3] A. Giordana and C. Sale. Learning structured concepts using genetic algorithms. In D. Sleeman and P. Edwards, editors, *Proceedings of the 9th International Workshop on Machine Learning*, pages 169–178. Morgan Kaufmann, 1992.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, Reading, MA, 1989.

[5] J. Hekanaho. Dogma: A ga-based relational learner. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446 of *Lecture Notes in Artificial Intelligence*, pages 205–214. Springer-Verlag, 1998.

[6] J.H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[7] Claire J. Kennedy and Christophe Giraud-Carrier. An evolutionary approach to concept learning with structured data. In *Proceedings of the fourth International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 1–6. Springer Verlag, April 1999.

[8] Y. Kodratoff and R. Michalski. Research in machine learning: Recent progress, classification of methods and future directions. In Y. Kodratoff and R. Michalski, editors, *Machine learning: an artificial intelligence approach*, volume 3, pages 3–30. Morgan Kaufman, San Mateo, CA, 1990.

[9] J. R. Koza. *Genetic Programming*. MIT Press, Cambridge, MA, 1991.

[10] K. S. Leung and M. L. Wong. Genetic logic programming and applications. *IEEE Expert*, 10(5):68–76, 1995.

[11] R.S. Michalski. Pattern recognition as rule-guided inductive inference. In *Proceedings of IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 349–361, 1980.

[12] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

[13] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.

[14] S-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag, Berlin, 1997. LNAI 1228.

[15] G.D. Plotkin. A note on inductive generalisation. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.

[16] A. Varšek. *Inductive Logic Programming with Genetic Algorithms*. PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Ljubljana, Slovenia, 1993. (In Slovenian).

New Conditions for the Existence of Least Generalizations under Relative Subsumption

Akihiro Yamamoto

Faculty of Technology and MemeMedia Laboratory, Hokkaido University
and
"Information and Human Activity", PRESTO, JST
MemeMedia Laboratory, N 13 W 8, Sapporo 060-8628 JAPAN
yamamoto@meme.hokudai.ac.jp

Abstract. Least common generalization under relative subsumption (LGRS) is a fundamental concept in Inductive Logic Programming. In this paper we give several new conditions for the existence of LGRSs. In previous researches the existence of LGRSs was guaranteed when a background theory is logically equivalent to conjunction of finitely many ground literals. Each of our conditions allows a background theory to have clauses with variables in it. The conditions are obtained using the bottom method (or the bottom generalization method), with which any clause subsuming a positive example relative to a background theory can be derived. We also compare the conditions with those for the existence of relative least generalizations under generalized subsumption (LGGSs).

1 Introduction

The relative subsumption relation was defined by Plotkin [12, 13]. The subsumption relation is well-known in the area of automated theorem proving, and the relative subsumption relation is one of its extensions. If a clause H subsumes a clause E relative to a background theory B, $H \wedge B$ logically implies E, but the converse does not hold in general. In spite of this semantical disadvantage, relative subsumption is more useful than logical implication because we showed in [18, 19] that every H subsuming E relative to B can be obtained with an extension of the execution method of logic programs. The method is called the *bottom method* (or the *bottom generalization method*). The fundamental idea of the bottom method is from Muggleton's *inverse entailment* [7] [1]. We also showed that Plotkin's SB-resolution is again an extension of execution of logic programs, with which we can examine whether or not H subsumes E relative to B. Based on relative subsumption Plotkin gave a bottom-up method for induction while Muggleton developed a top-down method based on inverse entailment. These facts tell that relative subsumption is a fundamental concept independent of the direction of hypothesis search.

In this paper we treat the case when many clauses $E_1, E_2, \ldots, E_n$ are given as positive examples and a clause H which subsumes all of E_i's relative to B is

[1] In [17, 18] the bottom method was not well distinguished from *inverse entailment*.

J. Cussens and A. Frisch (Eds.): ILP 2000, LNAI 1866, pp. 253–264, 2000.

looked for. Such H is called a *common generalization under relative subsumption* of $E_1, \ldots, E_n$. A least common generalization under relative subsumption is called an *LGRS* in this paper. The aim of our research is to clarify the logical properties of LGRSs and make them more useful in designing ILP systems.

Plotkin [12] showed that an LGRS of a finite set of clauses exists when a background theory B is a finite set of ground literals. Under this condition no background theory B has any clause with variables in it. In terminology of relational database, B can be a set of tuples but cannot be any view obtained with relational operations. This is quite inconvenient when we apply LGRS-based ILP methods to knowledge discovery from databases.

One previous approach to overcome the problem was seen in the GOLEM system [8]. In GOLEM B can be a set of definite clauses because it is replaced with its least Herbrand model $\mathrm{M}(B)$. However, since $\mathrm{M}(B)$ is neither finite nor logically equivalent to B in general, there is no proof that GOLEM always derives LGRSs. In [20] we gave a condition under which B is logically equivalent to $\mathrm{M}(B)$ and $\mathrm{M}(B)$ is finite, but B could not have any clause with variables under the condition.

In this paper we give some new conditions for the existence of LGRSs of clauses. Our condition allows a background theory to have clauses with variables in it. We derive them by using the results on the relation between relative subsumption and the bottom method.

With referring the conditions we also show some conditions for existing *least generalizations under generalized subsumption* (*LGGSs*). LGGS, which was defined by Buntine [3], is quite similar to LGGS. They are not well distinguished in many cases and called relative least general generalization (RLGG). The similarity derives the conditions for LGGSs from those for LGRSs.

This paper is organized as follows: After preparing some definitions and notations in Section 2, we give some formal definition of LGRS in Section 3. We also explain some basic conditions for the existence of LGRSs. In Section 4 we give some fundamental results on relative subsumption and the bottom method, and by using them we give new conditions for the existence of LGRSs. In Section 5 we compare our results with the conditions for the existence of LGGSs, and we give conclusions in Section 6.

2 Preliminaries

We assume that readers are familiar with fundamental terminology and concepts in first-order logic, clausal logic, logic programming, and ILP.

In this paper, we distinguish function symbols and constant symbols. We assume that every *function symbol* has one or more arguments while a *constant symbol* has no argument. A *clause* is the universal closure of a disjunction of finitely many literals

$$C = \forall(A_1 \vee A_2 \vee \ldots \vee A_n \vee \neg B_1 \vee \neg B_2 \vee \ldots \vee \neg B_m),$$

where each of A_i's and B_j's is an atom and $n, m \geq 0$. The symbol $\forall$ represents that each variable in $A_1, A_2, \ldots, B_1, B_2, \ldots$ is universally quantified. The clause

is called a *definite clause* if $n = 1$, and a *goal clause* if $n = 0$. The clause C is represented as

$$C = A_1; A_2; \ldots; A_n \leftarrow B_1, B_2, \ldots, B_m.$$

We define $C^+ = A_1; \ldots; A_n \leftarrow$ and $C^- = \leftarrow B_1, \ldots, B_m$ and call them the *head* and the *body* of C, respectively.

For a clause C, a substitution σ_C is defined which replaces each variable X in S with an new constant symbol c_X. The index C of σ_C is sometimes omitted when it is easily conjectured from the context. For a set S of literals we define

$$\overline{S} = \{\neg L \mid L \in S\}.$$

For a set of formulae $S = \{F_1, F_2, \ldots, F_n\}$, we define

$$\bigwedge S = F_1 \wedge F_2 \wedge \ldots \wedge F_n, \text{ and}$$
$$\bigvee S = F_1 \vee F_2 \vee \ldots \vee F_n.$$

A *clausal sentence* is a conjunction of finitely many clauses. A clausal sentence consisting of definite clauses is called a *definite program*. The *least Herbrand model* of a definite program P is denoted by $\mathrm{M}(P)$. The well-known operator T_P in logic programming is sometimes called the *covering operator* in ILP. Herbrand interpretations $T_P \uparrow n$ for $n = 0, 1, \ldots$ and $T_P \uparrow \omega$ is defined as follows:

$$\begin{aligned} &T_P \uparrow 0 = \emptyset, \\ &T_P \uparrow n = T_P \uparrow (n-1) \qquad \text{for } n = 1, 2, \ldots, \\ &T_P \uparrow \omega = \bigcup_{n \geq 0} T_P \uparrow n \end{aligned}$$

It holds that $\mathrm{M}(P) = T_P \uparrow \omega$.

We write $F_1 \vdash F_2$ if a formula F_2 is provable from a formula F_1 with a complete proof procedure. When F_1 is a clausal sentence and F_2 is a clause, a proof procedure whose inference rules are resolution with factoring and subsumption is known to be complete [6].

3 Least Generalization under Relative Subsumption

Let $\mathcal{L}$ be a first-order language. In the following discussion we assume a sub-language $\mathcal{L}_{\mathrm{H}}$ of $\mathcal{L}$ and a formula B of $\mathcal{L}$. The sub-language $\mathcal{L}_{\mathrm{H}}$ is called a *hypothetical language*, and each of its formulae is a *hypothesis*. The formula B is called a *background theory*. A *positive example* (or an *example*, for short) should be given as a formula in $\mathcal{L}_{\mathrm{H}}$.

Our discussion is given with comparing the following two cases:

- B is a clausal sentence and $\mathcal{L}_{\mathrm{H}} = \mathrm{C}(\mathcal{L})$, where $\mathrm{C}(\mathcal{L})$ denotes the set of clauses in $\mathcal{L}$.

– B is a logic program and $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$, where $\mathrm{D}(\mathcal{L})$ denotes the set of definite clauses in $\mathcal{L}$.

We firstly define generalizations and least generalizations relative to a background theory.

Definition 1. Let B be a background theory, and $\geq$ a partial order relative to B on the set of formulae in $\mathcal{L}_{\mathrm{H}}$. If $\geq$ satisfies

$$H \geq E \implies B \wedge H \vdash E,$$

$\geq$ is called a *generalization relation relative to* B.

Definition 2. Let $\geq$ be a generalization relation relative to B, and S be a set of hypotheses. A hypothesis H is a *common generalization* of S under $\geq$ in $\mathcal{L}_{\mathrm{H}}$ if $H \geq E$ for every E in S. A hypothesis K is a *least common generalization* (or *least generalization*) of S under $\geq$ in $\mathcal{L}_{\mathrm{H}}$ if K is a common generalization of S under $\geq$ and $H \geq K$ for any common generalization H of S under $\geq$ in $\mathcal{L}_{\mathrm{H}}$.

In this paper we use the word "generalization" to mean a common generalization with it. Readers should take care of the difference between "a generalization relation" and "a generalization."

Definition 3. Let H and E be clauses. We say that H *subsumes* E and write $H \succeq E$ if there is a substitution θ such that every literal in $H\theta$ occurs also in E.

Subsumption is often called θ*-subsumption*. Relative subsumption is a generalization relation in the case when $\mathcal{L}_{\mathrm{H}} = \mathrm{C}(\mathcal{L})$ and B is a clausal sentence in $\mathcal{L}$.

Definition 4 ([13]). Let H and E be clauses and B be a clausal sentence. We say that H *subsumes* E *relative to* B, and write $H \succeq_B E$ [2] if H subsumes such a clause F that $B \vdash \forall(m(E) \leftrightarrow m(F))$.

The expression $m(C)$ denotes the *matrix* of a clause C, which is the formula obtained by deleting the universal quantifier $\forall$ from C. The following propositions give another definitions of subsumption and relative subsumption.

Proposition 1. *For two clauses* H *and* E $H \succeq E$ *if and only if there is a substitution* θ *such that* $\vdash \forall(m(H\theta) \rightarrow m(E))$.

Proposition 2 ([13]). *Let* H *and* E *be clauses and* B *be a clausal sentence. Then* $H \succeq_B E$ *if and only if there is a substitution* θ *such that* $B \vdash \forall(m(H\theta) \rightarrow m(E))$.

[2] We sometimes write the expression as $H \succeq_{\mathrm{P}} E\ (B)$, as in [19], in order to stress that we adopt Plotkin's definition.

From the propositions we can easily know that subsumption relative to a tautology coincides with the subsumption relation.

Least generalization under relative subsumption is abbreviated to *LGRS*. *Least generalization under subsumption* is abbreviated to *LGS*. In this paper the LGS of a finite set of clauses $S = \{E_1, \ldots, E_n\}$ is denoted by $\mathrm{lgs}(E_1, \ldots, E_n)$.

Plotkin proved that $\mathrm{lgs}(E_1, \ldots, E_n)$ exists by giving an algorithm computing it [11]. The algorithm uses the following lemma repeatedly.

Lemma 1 ([11]). *Let E_1 and E_2 be clauses and*

$$M = \left\{ (L_1, L_2) \;\middle|\; \begin{array}{l} L_1 \text{ appears in } E_1,\ L_2 \text{ appears in } E_2 \text{ and} \\ \text{they have the same sign and predicate symbol} \end{array} \right\}.$$

Then it holds that

$$\mathrm{lgs}(E_1, E_2) = \bigvee \{ \mathrm{lca}(L_1, L_2) \mid (L_1, L_2) \in M \},$$

where $\mathrm{lca}(L_1, L_2)$ is the least common anti-instance of literals L_1 and L_2.

The algorithm computing $\mathrm{lca}(L_1, L_2)$ was given in [5, 11, 14].

Now we explain some known results on the existence of LGRSs. At first we must stress that a generalization is defined after fixing a hypothetical language $\mathcal{L}_{\mathrm{H}}$, a background theory B, and a generalization relation $\geq$. Since we consider the relative subsumption relation $\succeq_B$ as $\geq$, we must fix $\mathcal{L}_{\mathrm{H}}$ and B.

Example 1 ([9]). Let $B = \{p(a, X) \leftarrow, p(b, X) \leftarrow\}$ and that $\mathcal{L}_{\mathrm{H}} = \mathrm{C}(\mathcal{L})$. If $\mathcal{L}$ has at least one function symbol, then $S = \{q(a) \leftarrow, q(b) \leftarrow\}$ has no LGRS in $\mathcal{L}_{\mathrm{H}}$. The proof is shown in the text book by Nienhuys-Cheng and de Wolf [10].

The following theorem gives a condition under which any finite set of clauses has an LGRS. Note that $\neg B$ is a clause if B is a conjunction of ground literals. The language $\mathcal{L}$ may have function symbols.

Theorem 1 ([12]). *Let $\mathcal{L}_{\mathrm{H}}$ be $\mathrm{C}(\mathcal{L})$ and B be a conjunction of ground literals. Then any finite set of clauses $S = \{E_1, \ldots, E_n\}$ has an LGRS in $\mathrm{C}(\mathcal{L})$ which is equivalent to $\mathrm{lgs}(E_1 \vee \neg B, \ldots, E_n \vee \neg B)$.*

By Lemma 1 the LGS of two definite clauses E_1 and E_2 exists in $\mathrm{D}(\mathcal{L})$ if and only if ${E_1}^+$ and ${E_2}^+$ have the same predicate symbol. From this remark we can revise Theorem 1 for the case $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$.

Theorem 2. *Let $\mathcal{L}_{\mathrm{H}}$ be $\mathrm{D}(\mathcal{L})$ and B be a conjunction of ground atoms. Let $S = \{E_1, \ldots, E_n\}$ be a finite set of definite clauses such that all ${E_i}^+$ have the same predicate symbol. Then $\mathrm{lgs}(E_1 \vee \neg B, \ldots, E_n \vee \neg B)$ is the LGRS of S in $\mathrm{D}(\mathcal{L})$.*

4 New Conditions for the Existence of LGRSs

4.1 Fundamental Results on Relative Subsumption

In order to give some new conditions for the existence of LGRSs, we explain some procedural aspects of relative subsumption, which we showed in our previous works [4, 17, 18, 19, 21]

Definition 5. For a clause E and a clausal sentence B, we define the *bottom set* of E under B as

$$\mathrm{Bot}(E, B) = \{L \mid L \text{ is such a ground literal that } B \wedge \neg(E\sigma) \vdash \neg L\}.$$

Theorem 3 ([18, 19]). *A clause H subsumes a clause E relative to B if and only if $B \vdash E$ or H subsumes a clause F which is the disjunction of some literals in* $\mathrm{Bot}(E, B)$.

When $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$, each element in the bottom set can be derived with SOLDR-derivation and construction of the least Herbrand model of a definite program.

Definition 6. Let P be a definite program, G is a goal and R is a computation rule for selecting an atom from a goal. A sequence of quadruplets $(G_i, F_i, \theta_i, C_i)$ $(i = 0, 1, \ldots, n)$ is an *SOLDR-derivation* if it satisfies the following conditions:

1. For every $i = 0, 1, \ldots, n$, G_i and F_i are goal clauses and θ_i is a substitution. C_i is a clause in P whose variables are standardized apart.
2. $G_0 = G, F_0 = G_n = \square$ where $\square$ is an empty clause
3. If, for every $i = 0, 1, \ldots, n-1$, $G_i = \leftarrow A_1, \ldots, A_m$, $F_i = \leftarrow B_1, \ldots, B_h$, $(h \geq 0)$ and A_j is the atom selected from G according to R, then one of the following three holds:
   (a) (Resolution)
   θ_i is the mgu of C_i^+ and A_j $G_{i+1} = (\leftarrow A_1, \ldots, A_{j-1}, B_1, \ldots, B_k, A_{j+1}, \ldots, A_m)\theta_i$, and $F_{i+1} = F_i\theta_i$.
   (b) (Addition atoms to consequences)
   $F_i = \square$, $G_{i+1} = \leftarrow A_1, \ldots, A_{j-1}, A_{j+1}, \ldots, A_m$, and $F_{i+1} = \leftarrow A_j$.
   (c) (Reduction of consequences)
   $F_i = \leftarrow B$, θ_i is the mgu of A_j and B, $G_{i+1} = (\leftarrow A_1, \ldots, A_{j-1}, A_{j+1}, \ldots, A_m)\theta_i$, and $F_{i+1} = \leftarrow B\theta_i$.

The F_n of the last quadruplet is called the *consequence* of the SOLDR-derivation.

For a definite program P and a goal clause G, we define a set $SOLDR(P, G)$ of ground atoms with SOLDR-derivations as

$$SOLDR(P, G) = \left\{ \neg A \;\middle|\; \begin{array}{l} A \text{ is a ground atom such that } \leftarrow A \text{ is subsumed} \\ \text{by a goal } F \text{ which is the consequence of some} \\ \text{SOLDR-derivation from } (P, G) \end{array} \right\}.$$

Theorem 4 ([17, 21]). *Let B be a definite program and E be a definite clause. Then it holds that*

$$\mathrm{Bot}(E, B) = \overline{SOLDR(B \wedge \neg E^-\sigma_E, \neg E^+\sigma_E) \cup M(B \wedge \neg E^-\sigma_E)}.$$

4.2 Function-Free Languages

In the case that $\mathcal{L}$ has no function symbol, $\mathrm{Bot}(E, B)$ is a finite set of ground literals and we can define a clause

$$\mathrm{bot}(E, B) = \bigvee \mathrm{Bot}(E, B)$$

with calling it the *bottom clause* of E under B. We obtain the following corollary from Theorem 3.

Corollary 1. *Assume that $\mathcal{L}$ has no function symbol. Then a clause H subsumes a clause E relative to B if and only if $B \vdash E$ or $H \succeq \mathrm{bot}(E, B)$.*

Theorem 5. *Assume that $\mathcal{L}$ has no function symbol. Let B is an arbitrary clausal sentence. Then any finite set S of clauses has an LGRS in $\mathrm{C}(\mathcal{L})$.*

Proof. If $B \vdash E$, $H \succeq_B E$ for any H. Let S' be the set obtained by removing from S any E such that $B \vdash E$. If S' is empty, any tautological clause is the LGRS of S. Otherwise, let $S' = \{E_1, \ldots, E_n\}$. Then

$$K = \mathrm{lgs}(\mathrm{bot}(E_1, B), \ldots, \mathrm{bot}(E_n, B))$$

is an LGRS of S' in $\mathrm{C}(\mathcal{L})$ by Corollary 1. From the construction of S', K is also an LGRS of S in $\mathrm{C}(\mathcal{L})$.

In the case when $\mathcal{L}$ has no function symbol, it is decidable whether or not $B \vdash E$, and the clause $\mathrm{bot}(E_i, B)$ $(i = 1, \ldots, n)$ is also computable. Therefore the proof of Theorem 5 gives an algorithm computing an LGRS of S in $\mathrm{C}(\mathcal{L})$.

Example 2. Let us consider a background theory

$$B_{family} = \bigwedge \left\{ \begin{array}{l} p(a, b) \leftarrow \\ p(b, c) \leftarrow \\ p(d, e) \leftarrow \\ p(e, f) \leftarrow \\ m(a) \leftarrow \\ m(d) \leftarrow \\ gf(X, Y) \leftarrow f(X, Z), p(Z, Y) \end{array} \right\}$$

and a set of clauses

$$S_{gf} = \left\{ \begin{array}{l} E_{gf1} = gf(a, c) \leftarrow \\ E_{gf2} = gf(d, f) \leftarrow \end{array} \right\}.$$

The predicate symbols p, m, f, and gf represent relations of is_a_parent_of, is_male, is_the_father, and is_a_grandfather, respectively.

We compute an LGRS of S_{gf} in $\mathrm{C}(\mathcal{L})$. At first we compute the bottom clause of each of E_{gf1} and E_{gf2}. The positive literals in each bottom clause can be derived with SOLDR-derivations. The negative literals are derived with

computing $T_{B_{family}} \uparrow \omega$. Since $\mathcal{L}_{\mathrm{H}}$ is assumed to be $\mathrm{C}(\mathcal{L})$, we get the following bottom clauses:

$$\mathrm{bot}(E_{gf1}, B_{family}) = gf(a,c); f(a,b) \leftarrow p(a,b), p(b,c), p(d,e), p(e,f), m(a), m(d)$$
$$\mathrm{bot}(E_{gf2}, B_{family}) = gf(d,f); f(d,e) \leftarrow p(a,b), p(b,c), p(d,e), p(e,f), m(a), m(d)$$

According to the proof in Theorem 5, an LGRS of S_{gf} in $\mathrm{C}(\mathcal{L})$ is obtained by computing

$$\mathrm{lgs}(\mathrm{bot}(E_{gf1}, B_{family}), \mathrm{bot}(E_{gf2}, B_{family})).$$

After applying the reduction method [11] to the obtained clause, we finally obtain the following clause as an LGRS:

$$gf(X,Y); f(X,Z) \leftarrow p(X,Z), p(Z,Y), p(a,b), p(b,c), p(d,e), p(e,f), m(X), m(a), m(d).$$

If $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$ and B is a definite program, the conclusion of Theorem 5 does not hold in general.

Example 3. Let $B_{pqr} = (p(X) \leftarrow q(X)) \wedge (p(X) \leftarrow r(X))$ and $S_p = \{p(a) \leftarrow, q(b) \leftarrow\}$. Then we get the following bottom clauses:

$$\mathrm{bot}(p(a) \leftarrow, B_{pqr}) = p(a); q(a); r(a) \leftarrow$$
$$\mathrm{bot}(q(b) \leftarrow, B_{pqr}) = q(b); r(b) \leftarrow$$

If an LGRS of S_p in $\mathrm{D}(\mathcal{L})$ exists, it should be either of $H_q = q(X) \leftarrow$ and $H_r = r(X) \leftarrow$. It is easy to check that neither $H_q \succeq_{B_{pqr}} H_r$ nor $H_r \succeq_{B_{pqr}} H_q$ holds.

Referring the remark given before Theorem 2, we know that we need more complicated conditions for the existence of LGRSs in the case $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$. We give one of such conditions.

Theorem 6. *Assume that $\mathcal{L}$ has no function symbol. Let B be a definite program and $\mathcal{L}_{\mathrm{H}}$ be $\mathrm{D}(\mathcal{L})$. Then a set of definite clauses $S = \{E_1, \ldots, E_n\}$ has an LGRS in $\mathrm{D}(\mathcal{L})$ if the following three are satisfied:*

1. *$B \not\models E_i$ for $i = 1, 2, \ldots, n$.*
2. *There is a common predicate symbol p for $i = 1, 2, \ldots, n$ which occurs in exactly one literal in $SOLDR(B \wedge \neg({E_i}^-\sigma), \neg({E_i}^+\sigma))$.*
3. *For each pair of E_i and E_j $(i \neq j)$, no predicate symbol other than p occurs in $SOLDR(B \wedge \neg({E_i}^-\sigma), \neg({E_i}^+\sigma))$ and $SOLDR(B \wedge \neg({E_j}^-\sigma), \neg({E_j}^+\sigma))$.*

Proof. Let H be a definite clause such that $H \succeq_B E_i$. At first we do not mind the condition $\mathcal{L}_{\mathrm{H}} = \mathrm{C}(\mathcal{L})$. Then $H \succeq \mathrm{bot}(E_i, B)$ from Theorem 5. Let p be the predicate symbol satisfying the second condition of the theorem, and $p(t_{i1}, \ldots, t_{im})$ is in $SOLDR(B \wedge \neg({E_i}^-\sigma), \neg({E_i}^+\sigma))$. If $H \succeq_B E_i$ and $H \succeq_B E_j$ $(i \neq j)$, the predicate symbol of H^+ must be p from the third condition. Let $L_{i1}, L_{i2}, \ldots, L_{ih_i}$ be all negative literals in $\mathrm{bot}(E_i, B)$ and

$$K_i = p(t_{i1}, \ldots, t_{im}) \vee L_{i1} \vee L_{i2} \vee \ldots \vee L_{ih_i}$$

Then $\mathrm{lgs}(K_1, K_2, \ldots, K_n)$ is an LGRS of S in $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$.

Example 4. Let $B_{tsu} = (t(X) \leftarrow u(X)) \wedge (s(X) \leftarrow u(X))$ and $S_{ts} = \{t(a) \leftarrow, s(b) \leftarrow\}$. Since

$$SOLDR(B_{tsu}, \leftarrow t(a)) = \{t(a), u(a)\} \text{ and}$$
$$SOLDR(B_{tsu}, \leftarrow s(b)) = \{s(b), u(b)\},$$

the clause $\mathrm{lgs}(u(a) \leftarrow, u(b) \leftarrow) = u(X) \leftarrow$ is an LGRS of S_{ts} in $\mathrm{D}(\mathcal{L})$.

4.3 Linear Definite Clauses

The next condition for ensuring the existence of LGRSs is given by linear definite clauses and linear definite programs. They were originally introduced in the research on the execution of logic programs [2, 15] and then used in the research of Machine Learning [1, 16]. Linear definite clauses and linear definite programs can be independent of whether or not $\mathcal{L}$ has at least one function symbol.

In the following, the number of occurrences of function symbols and variables in an atom A is denoted by $|A|$. For a set S of atoms we define

$$S|_k = \{A \mid A \in S \text{ and } |A| \leq k\}.$$

Definition 7. ([2, 15]) A definite clause $A_0 \leftarrow A_1, \ldots, A_n$ is *linear* (or *weakly reducing*) if $|A_0\theta| \geq |A_i\theta|$ for any substitution θ and each A_i $(i = 1, 2, \ldots, n)$. A definite program P is *linear* (or *weakly reducing*) if P is a conjunction of linear definite clauses.

The set of linear definite clauses in $\mathcal{L}$ is denoted by $\mathrm{LD}(\mathcal{L})$. Note that any clause subsumed by a linear definite clause H is linear.

Let P be a linear definite program and let us consider a ground goal $G = \leftarrow A_0$. Then $|A_0| \geq |A|$ for any atom A in a goal in an SLD-derivation of G.

Lemma 2 ([4]). *Let B be a linear program and $\mathcal{L}_{\mathrm{H}}$ be $\mathrm{LD}(\mathcal{L})$. Then H subsumes E relative to B if and only if there is a ground atom*

$$A_0 \in \overline{SOLDR(B \wedge \neg(E^-\sigma), \neg(E^+\sigma))}$$

such that H subsumes

$$A_0 \leftarrow \bigwedge \mathrm{M}(B \wedge \neg(E^-\sigma))|_{|A_0|}.$$

By the above lemma we can give the next theorem in a similar way to Theorem 6.

Theorem 7. *Let B be a linear program and $\mathcal{L}_{\mathrm{H}}$ be $\mathrm{LD}(\mathcal{L})$. Then a set of linear definite clauses $S = \{E_1, \ldots, E_n\}$ has an LGRS in $\mathrm{LD}(\mathcal{L})$ if the following three are satisfied:*

1. *$B \not\models E_i$ for $i = 1, 2, \ldots, n$.*
2. *There is a common predicate symbol p for $i = 1, 2, \ldots, n$ which occurs in exactly one literal in $SOLDR(B \wedge \neg({E_i}^-\sigma), \neg({E_i}^+\sigma))$.*

3. *For each pair of* E_i *and* E_j $(i \neq j)$*, no predicate symbol other than* p *occurs in* $SOLDR(B \wedge \neg(E_i{}^-\sigma), \neg(E_i{}^+\sigma))$ *and* $SOLDR(B \wedge \neg(E_j{}^-\sigma), \neg(E_j{}^+\sigma))$.

When B is a linear logic program, we can compute $SOLDR(B \wedge \neg(E_i^-\sigma), \neg(E_i^+\sigma))$. This fact gives an algorithm for computing an LGRS under the three conditions of the theorem above. Moreover, since it is decidable whether or not $B \vdash E$, any clause E satisfying $B \vdash E$ from S can effectively removed from S as in the proof of Theorem 5.

The background theory B_{tsu} in Example 4 is a linear definite program, and both of the clauses in S_{ts} are linear definite clauses. The obtained LGRS $u(X) \leftarrow$ is also a linear definite clauses.

4.4 Least Generalizations under Generalized Subsumption

Generalized subsumption defined by Buntine [3] is another relative generalization relation which has often been used in research of ILP. Some results on *least generalizations under generalized subsumption* (*LGGS*, for short) can be compared with the results we showed in the previous section.

When $\mathcal{L}_{\mathrm{H}} = \mathrm{D}(\mathcal{L})$ and B is a definite program, generalized subsumption is defined as follows:

Definition 8 ([3]). Let H and E be definite clauses. If $T_H(M) \supseteq T_E(M)$ for every Herbrand model M of B, we say H *subsumes* E *w.r.t.* B and write $H \geq_B E$ [3].

As relative subsumption is characterized with bottom sets, generalized subsumption is characterized with saturated sets. A *saturated set* of a definite clause E under a definite program B is a set

$$\mathrm{Satu}(E, B) = \{E^+\sigma\} \cup \{\neg A \mid A \text{ is a ground atom such that } B \wedge \neg(E^-\sigma) \vdash A\},$$

where σ is a Skolemizing substitution for E. It was shown that H subsumes E w.r.t. B if and only if $B \vdash E$ or H subsumes a definite clauses F consisting of some literals in $\mathrm{Satu}(E, B)$. If $\mathrm{Satu}(E, B)$ is finite, we can define a clause $\mathrm{satu}(E, B) = \bigvee \mathrm{Satu}(E, B)$. The clause is called *saturant* of E under B.

The next theorem corresponds to Theorem 5 (see also Corollary 16.32 in [10]).

Theorem 8 ([3]). *Let* $\mathcal{L}$ *have no function symbol and* B *be a definite program. If* S *is a finite set of definite clauses all of which have the same predicate symbol in their head, there exists an LGGS of* S *in* $\mathrm{D}(\mathcal{L})$.

The LGGS is obtained in a similar way to the proof of Theorem 5. For LGGS we compute

$$K = \mathrm{lgs}(\mathrm{satu}(E_1, B), \ldots, \mathrm{satu}(E_n, B))$$

[3] As is in the case of relative subsumption, we sometimes write the expression as $H \succeq_{\mathrm{B}} E$ (B) where the index B indicates Buntine.

instead of $\mathrm{lgs}(\mathrm{bot}(E_1, B), \ldots, \mathrm{bot}(E_n, B))$. Here we have to refer the remark given before Theorem 2 again.

We can show a theorem for LGGS which is similar to Theorem 7.

Theorem 9. *Let B be a linear program and $\mathcal{L}_{\mathrm{H}} = \mathrm{LD}(\mathcal{L})$. The LGGS of a set S of hypotheses exists if $B \not\vdash E$ for every hypothesis E in S and all clauses in S have the same predicate symbol in their head.*

This theorem cannot be derived from Theorem 7.1 in [3](Theorem 16.29 in [10]).

5 Conclusion

In this paper we investigated logical properties of LGRSs and compared them with hose of LGGSs. An LGRS of a set of hypotheses S can be considered as the most appropriate in the hypotheses which are more general than the clauses in S. The appropriateness is defined from an algebraic view of the relative subsumption relation.

It has often been pointed out that computing LGRSs with Lemma 1 requires many computational resources. Various improvements have been invented for practical application of LGRS. One possible improvement is to abandon finding an exact LGRS, and to compute some of its approximations, as in Muggleton's system [7]. We would like to stress that finding an LGRS is a criterion for choosing an appropriate common generalization. Efficiency in computing a common generalization should be another criterion for appropriateness. Our future work is to give some theoretical foundations to computing a common generalization which is most appropriate from the view point of computational efficiency.

Acknowledgments

The author thanks Prof. Alan Frish for his comments which he gave me during his stay at the Meme Media Laboratory of Hokkaido University.

References

[1] Arikawa, S., Shinohara, T., and Yamamoto, A.: Learning Elementary Formal Systems, *Theoretical Computer Science*, Vol. 95, No. 1, pp. 97–113 (1992).
[2] Arimura, H.: Completeness of Depth-Bounded Resolution for Weakly Reducing Programs, in Nakata, I. and Hagiya, M. eds., *Software Science and Engineering (World Scientific Series in Computer Science Vol.31)*, pp. 227–245 (1991).
[3] Buntine, W.: Generalized Subsumption and its Applications to Induction and Redundancy, *Artificial Intelligence*, Vol. 36, pp. 149–176 (1988).
[4] Ito, K. and Yamamoto, A.: Finding Hypotheses from Examples by Computing the Least Generalization of Bottom Clauses, in *Proceedings of the First International Conference on Discovery Science (LNAI 1532)*, pp. 303–314, Springer (1998).
[5] Lassez, J.-L., Maher, M. J., and Marriott, K.: Unification Revisited, in (Minker, J. ed.)*Foundations of Deductive Databases and Logic Programming*, pp. 587–626, Morgan-Kaufman (1988).

[6] Lee, R. C. T.: *A completeness theorem and a computer program for finding theorems derivable from given axioms*, PhD Thesis, University of California, Berkeley, 1967.
[7] Muggleton, S.: Inverse Entailment and Progol, *New Generation Computing*, Vol. 13, pp. 245–286 (1995).
[8] Muggleton, S. and Feng, C.: Efficient Induction of Logic Programs, in S. Arikawa and S. Goto and S. Ohsuga and T. Yokomori, ed., *Proceedings of the First International Workshop on Algorithmic Learning Theory*, pp. 368–381, JSAI (1990).
[9] Niblett, T.: A Study of Generalization in Logic Programming, in D. Sleeman, ed., *Proceedings of the 3rd European Workingsessions on Learning (EWSL-88)*, pp. 131–138 (1988).
[10] Nienhuys-Cheng, S.-H. and de Wolf, R.: *Foundations of Inductive Logic Programming (LNAI 1228)*, Springer (1997).
[11] Plotkin, G. D.: A Note on Inductive Generalization, in *Machine Intelligence 5*, pp. 153–163, Edinburgh University Press (1970).
[12] Plotkin, G. D.: A Further Note on Inductive Generalization, in *Machine Intelligence 6*, pp. 101–124, Edinburgh University Press (1971).
[13] Plotkin, G. D.: *Automatic Methods of Inductive Inference*, PhD thesis, Edinburgh University (1971).
[14] Reynolds, J. C.: Transformational Systems and the Algebraic Structure of Atomic Formulas, in *Machine Intelligence 5*, pp. 135–151, Edinburgh University Press (1970).
[15] Shapiro, E.: Alternation and the Computational Complexity of Logic Programs, *J. Logic Programming*, Vol. 1, No. 1 (1984).
[16] Shinohara, T.: Inductive Inference of Monotonic Formal Systems from Positive Data, *New Generation Computing*, Vol. 8, pp. 371–384 (1991).
[17] Yamamoto, A.: Representing Inductive Inference with SOLD-Resolution, in *Proceedings of the IJCAI'97 Workshop on Abduction and Induction in AI*, pp. 59 – 63 (1997).
[18] Yamamoto, A.: Which Hypotheses Can Be Found with Inverse Entailment?, in *Proceedings of the Seventh International Workshop on Inductive Logic Programming (LNAI 1297)*, pp. 296 – 308 (1997). The extended abstract is in *Proceedings of the IJCAI'97 Workshop on Frontiers of Inductive Logic Programming*, pp.19–23 (1997).
[19] Yamamoto, A.: An Inference Method for the Complete Inverse of Relative Subsumption, *New Generation Computing*, Vol. 17, No. 1, pp. 99–117 (1999).
[20] Yamamoto, A.: Revising the Logical Foundations of Inductive Logic Programming Systems with Ground Reduced Programs, *New Generation Computing*, Vol. 17, No. 1, pp. 119–127 (1999).
[21] Yamamoto, A.: Using Abduction for Induction based on Bottom Generalization, in A. Kakas and P. Flach (eds.) *Abductive and Inductive Reasoning : Essays on their Relation and Integration*, pp. 99–117 (2000).

Author Index